Atacama
Large
Millimeter
Array

Agilent Technologies
E8257D PSG Signal Generators
Programming Guide

BEND-56.04.00.00-017-A-MAN

Version: A
Status: Released

2009-05-12

Prepared by:	**Organization:**	**Date:**
Agilent	NRAO	2009-05-12
Documentation PA Signature:		
Gretchen Stahlman	NRAO	
Approved by:		
Chris Langley Fabio Biancat Marchet	NRAO ESO	

Change Record

Version	Date	Affected section(s)	ICO #	Reason/Initiation/Remarks
A	2009-05-12	All	-	Initial Release

Programming Guide

(With Remote Operation and File Downloads)

Agilent Technologies
Signal Generators

This guide applies to the following signal generator models:

N5181A/82A MXG RF Signal Generators

N5183A MXG Microwave Analog Signal Generator

E4428C/38C ESG RF Signal Generators

E8663B RF Analog Signal Generator

E8257D/67D PSG Microwave Signal Generators

Due to our continuing efforts to improve our products through firmware and hardware revisions, signal generator design and operation may vary from descriptions in this guide. We recommend that you use the latest revision of this guide to ensure you have up-to-date product information. Compare the print date of this guide (see bottom of page) with the latest revision, which can be downloaded from the following websites:

http://www.agilent.com/find/mxg

http://www.agilent.com/find/esg

http://www.agilent.com/find/psg

http://www.agilent.com/find/e8663b

Agilent Technologies

Manufacturing Part Number: E8251-90355

Printed in USA

June 2008

© Copyright 2006-2008 Agilent Technologies, Inc.

Notice

The material contained in this document is provided “as is”, and is subject to being changed, without notice, in future editions.

Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied with regard to this manual and to any of the Agilent products to which it pertains, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or any of the Agilent products to which it pertains. Should Agilent have a written contract with the User and should any of the contract terms conflict with these terms, the contract terms shall control.

Trademarks

Throughout this book, trademarked names are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner with no intention of infringement of the trademark.

Contents

1 Getting Started with Remote Operation

Programming and Software/Hardware Layers . . . 2
Interfaces . . . 4
IO Libraries and Programming Languages . . . 5
- Agilent IO Libraries Suite . . . 6
- Windows NT and Agilent IO Libraries M (and Earlier) . . . 7
- Selecting IO Libraries for GPIB . . . 9
- Selecting IO Libraries for LAN . . . 9
- Programming Languages . . . 10

Using the Web Browser . . . 11
- Enabling the Signal Generator Web Server . . . 12

Preferences . . . 17
- Configuring the Display for Remote Command Setups (Agilent MXG) . . . 17
- Configuring the Display for Remote Command Setups (ESG/PSG/E8663B) . . . 18
- Getting Help (Agilent MXG) . . . 18
- Getting Help (ESG/PSG/E8663B) . . . 18
- Setting the Help Mode (ESG/PSG/E8663B) . . . 19

Troubleshooting . . . 19
Error Messages . . . 20
- Error Message File . . . 20
- Error Message Types . . . 21

2 Using IO Interfaces

Using GPIB . . . 23
- Installing the GPIB Interface . . . 23
- Set Up the GPIB Interface . . . 25
- Verify GPIB Functionality . . . 26
- GPIB Interface Terms . . . 27

GPIB Programming Interface Examples . . . 27
- Before Using the GPIB Examples . . . 27
- Interface Check using HP Basic and GPIB . . . 27
- Interface Check Using NI–488.2 and C++ . . . 28

Using LAN . . . 29
- Setting Up the LAN Interface . . . 30
- Setting up Private LAN . . . 34
- Verifying LAN Functionality . . . 34
- Using VXI–11 . . . 39
- Using Sockets LAN . . . 40
- Using Telnet LAN . . . 41
- Using FTP . . . 45

Contents

Using LXI . . . 47
Using RS–232 (ESG, PSG, and E8663B Only) . . . 58
Selecting IO Libraries for RS–232 . . . 59
Setting Up the RS–232 Interface . . . 60
Verifying RS–232 Functionality . . . 62
Character Format Parameters . . . 63
If You Have Problems . . . 63
RS–232 Programming Interface Examples . . . 63
Before Using the Examples . . . 63
Interface Check Using HP BASIC . . . 64
Interface Check Using VISA and C . . . 64
Queries Using HP Basic and RS–232 . . . 65
Queries for RS–232 Using VISA and C . . . 65
Using USB (Agilent MXG) . . . 66
Selecting I/O Libraries for USB . . . 67
Setting Up the USB Interface . . . 67

3 Programming Examples

Using the Programming Interface Examples . . . 69
Programming Examples Development Environment . . . 70
Running C++ Programs . . . 70
Running C# Examples . . . 71
Running Basic Examples . . . 71
Running Java Examples . . . 72
Running MATLAB Examples . . . 72
Running Perl Examples . . . 73
Using GPIB . . . 73
Installing the GPIB Interface Card . . . 73
GPIB Programming Interface Examples . . . 74
Before Using the GPIB Examples . . . 74
GPIB Function Statements (Command Messages) . . . 74
Interface Check using HP Basic and GPIB . . . 78
Interface Check Using NI-488.2 and C++ . . . 79
Interface Check for GPIB Using VISA and C . . . 80
Local Lockout Using HP Basic and GPIB . . . 81
Local Lockout Using NI-488.2 and C++ . . . 82
Queries Using HP Basic and GPIB . . . 84
Queries Using NI-488.2 and Visual C++ . . . 85
Queries for GPIB Using VISA and C . . . 87
Generating a CW Signal Using VISA and C . . . 89

Generating an Externally Applied AC-Coupled FM Signal Using VISA and C 91
Generating an Internal FM Signal Using VISA and C. 93
Generating a Step-Swept Signal Using VISA and C++ 95
Generating a Swept Signal Using VISA and Visual C++ 96
Saving and Recalling States Using VISA and C. 98
Reading the Data Questionable Status Register Using VISA and C 101
Reading the Service Request Interrupt (SRQ) Using VISA and C 105
Using 8757D Pass-Thru Commands (PSG with Option 007 Only) 109
LAN Programming Interface Examples 112
VXI-11 Programming. 112
VXI-11 Programming Using SICL and C++. 113
VXI-11 Programming Using VISA and C++. 114
Sockets LAN Programming and C 116
Queries for Lan Using Sockets 119
Sockets LAN Programming Using Java 139
Sockets LAN Programming Using PERL 141
RS-232 Programming Interface Examples (ESG/PSG/E8663B Only) 142
Before Using the Examples. 142
Interface Check Using HP BASIC 142
Interface Check Using VISA and C 143
Queries Using HP Basic and RS-232 145
Queries for RS-232 Using VISA and C 146

4 Programming the Status Register System

Overview 149
Overall Status Byte Register Systems 151
Status Register Bit Values 160
Example: Enable a Register 160
Example: Query a Register 160
Accessing Status Register Information 161
Determining What to Monitor 161
Deciding How to Monitor 161
Status Register SCPI Commands 163
Status Byte Group 167
Status Byte Register 168
Service Request Enable Register 168
Status Groups 169
Standard Event Status Group 170
Standard Operation Status Group 172
Baseband Operation Status Group 175

Contents

Data Questionable Status Group . . . 177
Data Questionable Power Status Group . . . 181
Data Questionable Frequency Status Group . . . 184
Data Questionable Modulation Status Group . . . 187
Data Questionable Calibration Status Group . . . 190
Data Questionable BERT Status Group . . . 193

5 Creating and Downloading Waveform Files

Overview of Downloading and Extracting Waveform Files . . . 198
Waveform Data Requirements . . . 199
Understanding Waveform Data . . . 199
Bits and Bytes . . . 199
LSB and MSB (Bit Order) . . . 200
Little Endian and Big Endian (Byte Order) . . . 200
Byte Swapping . . . 202
DAC Input Values . . . 202
2's Complement Data Format . . . 205
I and Q Interleaving . . . 205
Waveform Structure . . . 207
File Header . . . 207
Marker File . . . 207
I/Q File . . . 209
Waveform . . . 209
Waveform Phase Continuity . . . 209
Phase Discontinuity, Distortion, and Spectral Regrowth . . . 209
Avoiding Phase Discontinuities . . . 210
Waveform Memory . . . 212
Memory Allocation . . . 214
Memory Size . . . 216
Commands for Downloading and Extracting Waveform Data . . . 218
Waveform Data Encryption . . . 218
File Transfer Methods . . . 219
SCPI Command Line Structure . . . 220
Commands and File Paths for Downloading and Extracting Waveform Data . . . 220
FTP Procedures . . . 224
Creating Waveform Data . . . 227
Code Algorithm . . . 227
Downloading Waveform Data . . . 233
Using Simulation Software . . . 234
Using Advanced Programming Languages . . . 236

Loading, Playing, and Verifying a Downloaded Waveform . . . 239
Loading a File from Non–Volatile Memory . . . 239
Playing the Waveform . . . 240
Verifying the Waveform . . . 241
Building and Playing Waveform Sequences . . . 241
Using the Download Utilities . . . 242
Downloading E443xB Signal Generator Files . . . 243
E443xB Data Format . . . 244
Storage Locations for E443xB ARB files . . . 244
SCPI Commands . . . 246
Programming Examples . . . 246
C++ Programming Examples . . . 247
MATLAB Programming Examples . . . 268
Visual Basic Programming Examples . . . 275
HP Basic Programming Examples . . . 280
Troubleshooting Waveform Files . . . 287
Configuring the Pulse/RF Blank (Agilent MXG) . . . 288
Configuring the Pulse/RF Blank (ESG/PSG) . . . 288

6 Creating and Downloading User–Data Files

Overview . . . 290
Signal Generator Memory . . . 291
Memory Allocation . . . 293
Memory Size . . . 294
Checking Available Memory . . . 295
User File Data (Bit/Binary) Downloads (E4438C and E8267D) . . . 297
User File Bit Order (LSB and MSB) . . . 298
Bit File Type Data . . . 298
Binary File Type Data . . . 301
User File Size . . . 302
Determining Memory Usage for Custom and TDMA User File Data . . . 303
Downloading User Files . . . 306
Command for Bit File Downloads . . . 309
Commands for Binary File Downloads . . . 310
Selecting a Downloaded User File as the Data Source . . . 311
Modulating and Activating the Carrier . . . 312
Modifying User File Data . . . 312
Understanding Framed Transmission For Real–Time TDMA . . . 315
Real–Time Custom High Data Rates . . . 319
Pattern RAM (PRAM) Data Downloads (E4438C and E8267D) . . . 320

Contents

Understanding PRAM Files . . . 321
PRAM File Size . . . 324
SCPI Command for a List Format Download . . . 325
SCPI Command for a Block Data Download . . . 326
Selecting a Downloaded PRAM File as the Data Source . . . 329
Modulating and Activating the Carrier . . . 330
Storing a PRAM File to Non–Volatile Memory and Restoring to Volatile Memory . . . 330
Extracting a PRAM File . . . 330
Modifying PRAM Files . . . 332
FIR Filter Coefficient Downloads (N5162A, N5182A, E4438C and E8267D) . . . 333
Data Requirements . . . 333
Data Limitations . . . 333
Downloading FIR Filter Coefficient Data . . . 334
Selecting a Downloaded User FIR Filter as the Active Filter . . . 334
Save and Recall Instrument State Files . . . 336
Save and Recall SCPI Commands . . . 336
Save and Recall Programming Example Using VISA and C# . . . 337
User Flatness Correction Downloads Using C++ and VISA . . . 347
Data Transfer Troubleshooting (N5162A, N5182A, E4438C and E8267D Only) . . . 351
User File Download Problems . . . 351
PRAM Download Problems . . . 352
User FIR Filter Coefficient File Download Problems . . . 354

Documentation Overview

Installation Guide

- Safety Information
- Receiving the Instrument
- Environmental & Electrical Requirements
- Basic Setup
- Accessories
- Operation Verification
- Regulatory Information

User's Guide

- Signal Generator Overview
- Setting Preferences & Enabling Options
- Basic Operation
- Optimizing Performance
- Using Analog Modulation (Option UNT Only)
- Using Pulse Modulation (Option UNU Only)
- Basic Digital Operation - No BBG Option Installed
- Basic Digital Operation (Option 651/652/654)
- Adding Real-Time Noise to a Signal (Option 403)
- Real-Time Phase Noise Impairments (Option 432)
- Custom Digital Modulation (Option 431)
- Multitone and Two-Tone Waveform Generator (Option 430)
- Working in a Secure Environment
- Troubleshooting

Programming Guide

- Getting Started with Remote Operation
- Using IO Interfaces
- Programming Examples
- Programming the Status Register System
- Creating and Downloading Files
- Creating and Downloading User-Data Files

SCPI Reference	• SCPI Basics • Basic Function Commands • LXI System Commands • System Commands • Analog Modulation Commands • Arb Commands • Real-Time Commands • N5161A/62A/81A/82A SCPI Command Compatibility • N5183A SCPI Command Compatibility
Service Guide	• Troubleshooting • Replaceable Parts • Assembly Replacement • Post-Repair Procedures • Safety and Regulatory Information • Instrument History
Key Help[a]	• Key function description • Related SCPI commands

[a]Press the **Help** hardkey, and then the key for which you wish help.

1 Getting Started with Remote Operation

CAUTION Agilent does *not* recommend going backwards in firmware versions (loading older firmware versions into newer instruments) as hardware/firmware conflicts can result.

NOTE For the N5161A/62A, the softkey menus and features mentioned in this chapter are only available through the Web-Enabled MXG or via SCPI commands. Refer to "Using the Web Browser" on page 11 and to the *SCPI Command Reference*.

The MXG ATE blank front panel models, N5161A and N5162A signal generators, are part of the MXG instrument family and unless otherwise indicated, all references to the MXG are inclusive of the MXG ATE instruments.

LXI Class B Compliance Disclaimer

As of this product's firmware release in June 2008, LXI Class B Compliance Tests, using the new IEEE 1588-2008 Precision Time Protocol (PTP), were not available. This product provides the features of an LXI Class B instrument by adding LAN Triggering and Time Synchronization to its LXI Class C compliance. References to LXI Class B in the document and in the instrument menus and web pages do not insure LXI Class B compliance. This product will apply for LXI Class B Compliance when the LXI Consortium has approved its specification changes and can provide certified testing. For more information on the LXI standards, refer to *http://www.lxistandard.org/home*.

LXI-B is only available on instruments with serial prefixes ≥US/MY/SG4818.

- Programming and Software/Hardware Layers on page 2
- Interfaces on page 4
- IO Libraries and Programming Languages on page 5
- Using the Web Browser on page 11
- Preferences on page 17
- Error Messages on page 20

Programming and Software/Hardware Layers

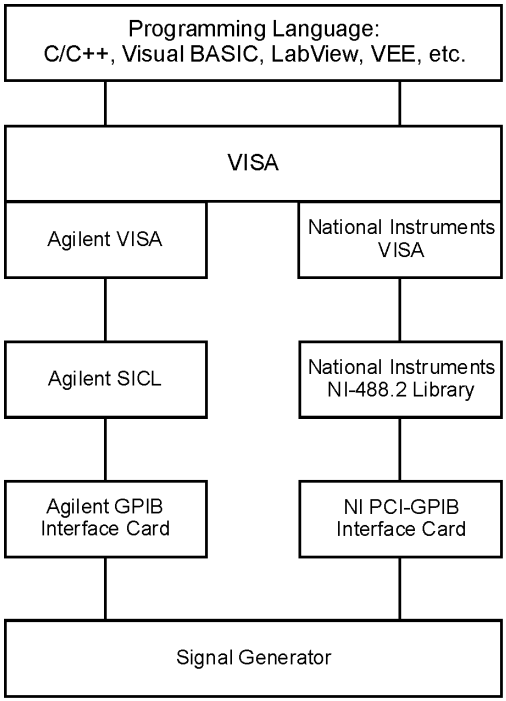
Agilent MXG, ESG, PSG, and E8663B signal generators support the following interfaces:

Instrument	Interfaces Supported
Agilent MXG	GPIB, LAN, and USB 2.0
Agilent E8663B[a]	GPIB, LAN, and ANSI/EIA232 (RS-232) serial connection
Agilent ESG	GPIB, LAN, and ANSI/EIA232 (RS-232) serial connection
Agilent PSG[a]	GPIB, LAN, and ANSI/EIA232 (RS-232) serial connection

a. The PSG and E8663B's **AUXILIARY INTERFACE** connector is compatible with ANSI/EIA232 (RS-232) serial connection but GPIB and LAN are recommended for making faster measurements and when downloading files. Refer to "Using RS–232 (ESG, PSG, and E8663B Only)" on page 58 and the *User's Guide*.

Use these interfaces, in combination with IO libraries and programming languages, to remotely control a signal generator. Figure 1-1 uses GPIB as an example of the relationships between the interface, IO libraries, programming language, and signal generator.

Figure 1-1 Software/Hardware Layers



Interfaces

GPIB

GPIB is used extensively when a dedicated computer is available for remote control of each instrument or system. Data transfer is fast because GPIB handles information in bytes with data transfer rates of up to 8 MBps. GPIB is physically restricted by the location and distance between the instrument/system and the computer; cables are limited to an average length of two meters per device with a total length of 20 meters.

For more information on configuring the signal generator to communicate over the GPIB, refer to "Using GPIB" on page 23.

LAN

Data transfer using the LAN is fast as the LAN handles packets of data. The single cable distance between a computer and the signal generator is limited to 100 meters (100Base-T and 10Base-T).

The Agilent MXG is capable of 100Base-T LAN communication. The ESG, PSG and E8663B are designed to connect with a 10Base-T LAN. Where auto-negotiation is present, the ESG, PSG, and E8663B can connect to a 100Base-T LAN, but communicate at 10Base-T speeds. For more information on LAN communication refer to *http://www.ieee.org*.

The following protocols can be used to communicate with the signal generator over the LAN:

- VXI-11 (recommended)
- Sockets
- TELNET
- FTP

The Agilent MXG supports LXI Class B[a] functionality. For more information on the LXI standards, refer to *http://www.lxistandard.org/home*.

For more information on configuring the signal generator to communicate over the LAN, refer to "Using LAN" on page 29.

RS-232[b] (ESG/PSG/E8663B Only)

RS-232 is an older method used to communicate with a single instrument; its primary use is to control printers and external disk drives, and connect to a modem. Communication over RS-232 is much slower than with GPIB, USB, or LAN because data is sent and received one bit at a time. It also requires that certain parameters, such as baud rate, be matched on both the computer and signal generator.

CAUTION For long strings of commands and waveform downloads, upgrading to Agilent IO Libraries 15.0 and above can *decrease* RS-232 performance, potentially resulting in an Error –310.

NOTE Because GPIB, LAN, and USB offer better communication performance, Agilent recommends that RS-232 only be used for interactive sessions or short commands.

For more information on configuring the signal generator to communicate over the RS-232, refer to "Using RS–232 (ESG, PSG, and E8663B Only)" on page 58.

USB (Agilent MXG Only)

- The rear panel Mini-B 5 pin connector is a device USB and can be used to connect a controller for remote operation.
- The Type-A front panel connector is a host USB and can be used to connect a mouse, a keyboard, or a USB 1.1/2.0 flash drive.

USB 2.0's 64 MBps communication speed is faster than GPIB (for data transfers, >1 KB) or RS-232. (For additional information, refer to the Agilent SICL or VISA User's Guide.) But, the latency for small transfers is longer.

For more information on connecting the signal generator to the USB, refer to the "Agilent IO Libraries Suite" on page 6 and the Agilent Connection Expert in the Agilent IO Libraries Help.

For more information on configuring the signal generator to communicate over the USB, refer to "Using USB (Agilent MXG)" on page 66.

a. LXI Class B Compliance testing using IEEE 1588-2008 not available at release.
b. The ESG, PSG, and E8663B's **AUXILIARY INTERFACE** connector is compatible with ANSI/EIA232 (RS-232) serial connection but GPIB and LAN are recommended for making faster measurements and when downloading files. Refer to "Using RS–232 (ESG, PSG, and E8663B Only)" on page 58 and the *User's Guide*.

IO Libraries and Programming Languages

The IO libraries is a collection of functions used by a programming language to send instrument commands and receive instrument data. Before you can communicate and control the signal generator, you must have an IO library installed on your computer. The Agilent IO libraries are included on an Automation-Ready CD with your signal generator and Agilent GPIB interface board, or they can be downloaded from the Agilent website: *http://www.agilent.com*.

CAUTION For long strings of commands and waveform downloads, upgrading to Agilent IO Libraries 15.0 and above can *decrease* RS-232 performance, potentially resulting in an Error –310.

NOTE To learn about using IO libraries with Windows XP or newer operating systems, refer to the Agilent IO Libraries Suite's help located on the Automation-Ready CD that ships with your signal generator. Other sources of this information, can be found with the Agilent GPIB interface board's CD, or downloaded from the Agilent website: *http://www.agilent.com*.

To better understand setting up Windows XP operating systems and newer, using PC LAN port settings, refer to Chapter 2.

Agilent IO Libraries Suite

The Agilent IO Libraries Suite replaces earlier versions of the Agilent IO Libraries. Agilent IO Libraries Suite does not support Windows NT. If you are using the Windows NT platform, you must use Agilent IO Libraries version M or earlier.

Windows 98 and Windows ME are not supported in the Agilent IO Libraries Suite version 14.1 and higher.

CAUTION The Agilent MXG's USB interface requires Agilent IO Libraries Suite 14.1 or newer. For more information on connecting instruments to the USB, refer to the Agilent Connection Expert in the Agilent IO Libraries Help.

For long strings of commands and waveform downloads, upgrading to Agilent IO Libraries 15.0 and above can *decrease* RS-232 performance, potentially resulting in an Error –310.

NOTE The signal generator ships with an Automation-Ready CD that contains the Agilent IO Libraries Suite 14.0 for users who use Windows 98 and Windows ME. These older systems are no longer supported.

Once the libraries are loaded, you can use the Agilent Connection Expert, Interactive IO, or VISA Assistant to configure and communicate with the signal generator over different IO interfaces. Follow instructions in the setup wizard to install the libraries.

Windows NT and XP are registered trademarks of Microsoft Corporation.

NOTE Before setting the LAN interface, the signal generator must be configured for VXI-11 SCPI. Refer to "Configuring the VXI–11 for LAN (Agilent MXG)" on page 30 or "Configuring the VXI–11 for LAN (ESG/PSG/E8663B)" on page 31.

Refer to the Agilent IO Libraries Suite Help documentation for details about this software.

Windows NT and Agilent IO Libraries M (and Earlier)

NOTE Windows NT is not supported on Agilent IO Libraries 14.0 and newer.

The following sections are specific to Agilent IO Libraries versions M and earlier and apply only to the Windows NT platform.

For additional information on older versions of Agilent IO libraries, refer to the Agilent Connection Expert in the Agilent IO Libraries Help. The Agilent IO libraries are included with your signal generator or Agilent GPIB interface board, or they can be downloaded from the Agilent website: *http://www.agilent.com*.

Using IO Config for Computer-to-Instrument Communication with VISA (Automatic or Manually)

After installing the Agilent IO Libraries version M or earlier, you can configure the interfaces available on your computer by using the IO Config program. This program can setup the interfaces that you want to use to control the signal generator. The following steps set up the interfaces.

1. Install GPIB interface boards before running IO Config.

NOTE You can also connect GPIB instruments using the Agilent 82357A USB/GPIB Interface Converter, which eliminates the need for a GPIB card. For more information, go to *http://www.agilent.com/find/gpib*.

2. Run the IO Config program. The program automatically identifies available interfaces.
3. Click on the interface type you want to configure, such as GPIB, in the Available Interface Types text box.
4. Click the **Configure** button. Set the Default Protocol to AUTO.
5. Click **OK** to use the default settings.
6. Click **OK** to exit the IO Config program.

VISA Assistant

VISA is an industry standard IO library API. It allows the user to send SCPI commands to instruments and to read instrument data in a variety of formats. You can use the VISA Assistant, available with the Agilent IO Libraries versions M and earlier, to send commands to the signal generator. If the interface you want to use does not appear in the VISA Assistant then you must manually configure the interface. See the Manual VISA Configuration section below. Refer to the VISA Assistant Help menu and the Agilent VISA User's Manual (available on Agilent's website) for more information.

VISA Configuration (Automatic)

1. Run the VISA Assistant program.
2. Click on the interface you want to use for sending commands to the signal generator.
3. Click the **Formatted I/O** tab.
4. Select **SCPI** in the **Instr. Lang.** section.

You can enter SCPI commands in the text box and send the command using the **viPrintf** button.

VISA Configuration (Manual)

Perform the following steps to use IO Config and VISA to manually configure an interface.

1. Run the **IO Config** Program.
2. Click on **GPIB** in the **Available Interface Types** text box.
3. Click the **Configure** button. Set the **Default Protocol** to **AUTO** and then click **OK** to use the default settings.
4. Click on **GPIB0** in the **Configured Interfaces** text box.
5. Click **Edit...**
6. Click the **Edit VISA Config...** button.
7. Click the **Add device** button.
8. Enter the GPIB address of the signal generator.
9. Click the **OK** button in this form and all other forms to exit the IO Config program.

Selecting IO Libraries for GPIB

The IO libraries are included with the GPIB interface card, and can be downloaded from the National Instruments website or the Agilent website. See also, "IO Libraries and Programming Languages" on page 5 for information on IO libraries. The following is a discussion on these libraries.

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is *not* recommended by Agilent Technologies.

VISA — VISA is an IO library used to develop IO applications and instrument drivers that comply with industry standards. It is recommended that the VISA library be used for programming the signal generator. The NI-VISA™ and Agilent VISA libraries are similar implementations of VISA and have the same commands, syntax, and functions. The differences are in the lower level IO libraries; NI-488.2 and SICL respectively. It is best to use the Agilent VISA library with the Agilent GPIB interface card or NI-VISA with the NI PCI-GPIB interface card.

SICL — Agilent SICL can be used without the VISA overlay. The SICL functions can be called from a program. However, if this method is used, executable programs will not be portable to other hardware platforms. For example, a program using SICL functions will not run on a computer with NI libraries (PCI-GPIB interface card).

NI-488.2 — NI-488.2 can be used without the VISA overlay. The NI-488.2 functions can be called from a program. However, if this method is used, executable programs will not be portable to other hardware platforms. For example, a program using NI-488.2 functions will not run on a computer with Agilent SICL (Agilent GPIB interface card).

Selecting IO Libraries for LAN

The TELNET and FTP protocols do not require IO libraries to be installed on your computer. However, to write programs to control your signal generator, an IO library must be installed on your computer and the computer configured for instrument control using the LAN interface.

The Agilent IO libraries Suite is available on the Automation-Ready CD, which was shipped with your signal generator. The libraries can also be downloaded from the Agilent website. The following is a discussion on these libraries.

Agilent VISA — VISA is an IO library used to develop IO applications and instrument drivers that comply with industry standards. Use the Agilent VISA library for programming the signal generator over the LAN interface.

SICL — Agilent SICL is a lower level library that is installed along with Agilent VISA.

NI-VISA is a registered trademark of National Instruments Corporation.

Programming Languages

Along with Standard Commands for Programming Instructions (SCPI) and IO library functions, you use a programming language to remotely control the signal generator. Common programming languages include:

- C/C++
- C#
- MATLAB® (MATLAB is a registered trademark of The MathWorks.)
- HP Basic
- LabView
- Java™ (Java is a U.S. trademark of Sun Microsystems, Inc.)
- Visual Basic® (Visual Basic is a registered trademark of Microsoft Corporation.)
- PERL
- Agilent VEE

For examples, using some of these languages, refer to Chapter 3.

Using the Web Browser

NOTE The Web-Enabled MXG ATE web page is titled: "Web-Enabled MXG", since the MXG ATE is part of the MXG signal generator family.

The instrument can be accessed through a standard web browser, when it is connected to the LAN. To access through the web browser, enter the instrument IP address as the URL in your browser.

The signal generator web page, shown at right and page 14, provides general information on the signal generator, FTP access to files stored on the signal generator, and a means to control the instrument using either a remote front-panel interface or SCPI commands. The web page also has links to Agilent's products, support, manuals, and website. For additional information on memory catalog access (file storing), and FTP, refer to the User's Guide and "Waveform Memory" on page 212 and for FTP, see "Using FTP" on page 45 and "FTP Procedures" on page 224.

The Web Server service is compatible with the Microsoft© Internet Explorer (6.0 and newer) web browser and operating systems Windows 2000, Windows XP, and newer. For more information on using the Web Server, refer to "Enabling the Signal Generator Web Server" on page 12.

The Agilent MXG supports LXI Class B* functionality. For more information on the LXI standards, refer to *http://www.lxistandard.org/home*.

*LXI Class B Compliance testing using IEEE 1588-2008 not available at release.

To operate the signal generator, click the keys.

Note: If you do *not* see this window, check to see if the window is hidden behind your browser window or your web browser settings are set to block pop-ups. To use this feature, you need to set your web browser to allow pop-ups for your instrument's IP address.

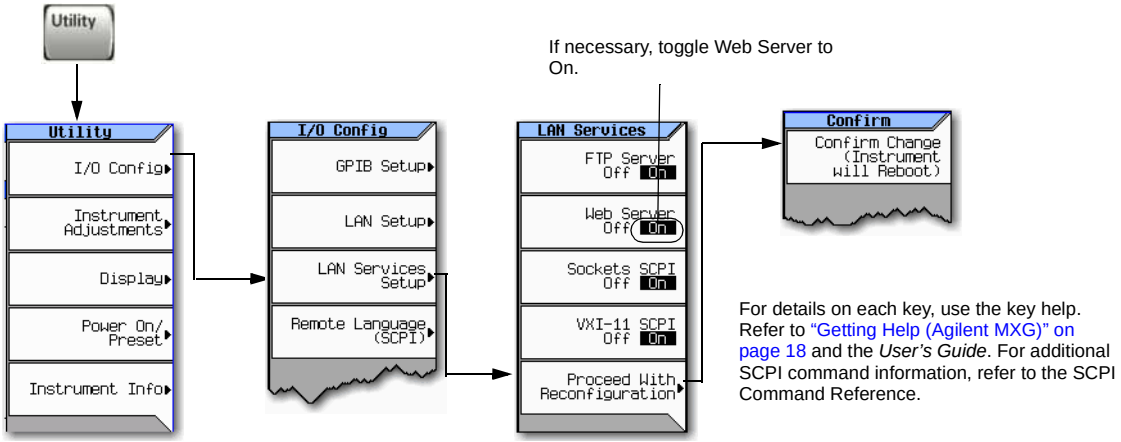
Microsoft is a registered trademark of Microsoft.

Enabling the Signal Generator Web Server

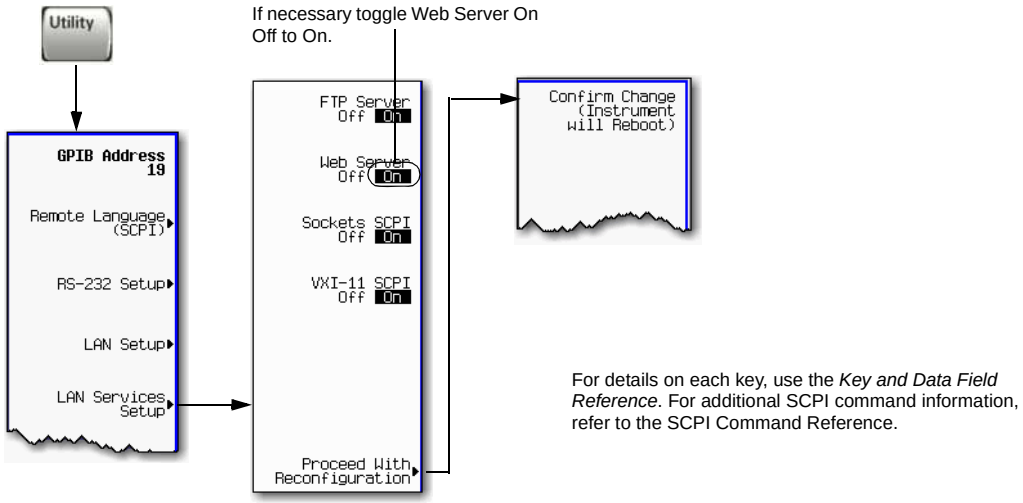
NOTE Javascript or Active Scripts must be enabled to use the web front panel controls.

1. Turn on the Web server as shown below.

Agilent MXG Web Server On


Utility
Utility
I/O Config
Instrument Adjustments
Display
Power On/ Preset
Instrument Info
I/O Config
GPIB Setup
LAN Setup
LAN Services Setup
Remote Language (SCPI)
LAN Services
FTP Server Off On
Web Server Off On
Sockets SCPI Off On
VXI-11 SCPI Off On
Proceed With Reconfiguration
If necessary, toggle Web Server to On.
Confirm
Confirm Change (Instrument will Reboot)
For details on each key, use the key help. Refer to "Getting Help (Agilent MXG)" on page 18 and the User's Guide. For additional SCPI command information, refer to the SCPI Command Reference.

ESG/PSG/E8663B Web Server On


Utility
If necessary toggle Web Server On Off to On.
GPIB Address 19
Remote Language (SCPI)
RS-232 Setup
LAN Setup
LAN Services Setup
FTP Server Off On
Web Server Off On
Sockets SCPI Off On
VXI-11 SCPI Off On
Proceed With Reconfiguration
Confirm Change (Instrument will Reboot)
For details on each key, use the Key and Data Field Reference. For additional SCPI command information, refer to the SCPI Command Reference.

2. Launch the PC or workstation web browser.

3. In the web browser address field, enter the signal generator's IP address. For example, *http://101.101.101.101* (where *101.101.101.101* is the signal generator's IP address).

   The IP (internet protocol) address can change depending on the LAN configuration (see "Using LAN" on page 29).

4. On the computer's keyboard, press **Enter**. The web browser displays the signal generator's homepage.
5. Click the Signal Generator Web Control menu button on the left of the page. The LXI password box is displayed on the computer. Refer to the *Web-Enabled MXG Help*.
6. Click **Submit**.
7. The front panel web page displays.

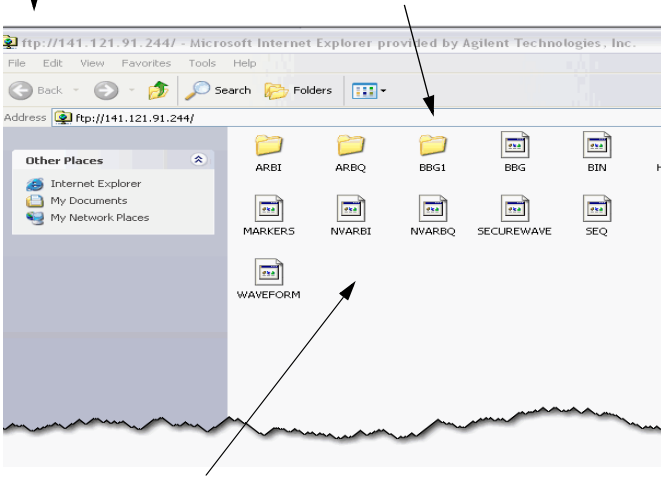
NOTE If you are experiencing problems with opening the signal generator's remote front panel web page, verify that the pop-up blocker is turned off on your web browser.

In some cases the Web-Enabled front panel may appear behind the main browser window, so you must move the browser window to see the Web-Enabled front panel.

To control the signal generator, either click the front panel keys or enter SCPI commands.

FTP enables the transfer of files between the instrument and a computer. The FTP access button provides drag-and-drop file capability.

The FTP access softkey opens to show the folders containing the signal generator's memory catalog files.



Use the FTP window to drag and drop files from the FTP page to your computer.

LAN Configuration System Defaults (Agilent MXG)

NOTE The instrument's LAN configuration system information can be found on the signal generator's homepage and on the signal generator. Refer to "Enabling the Signal Generator Web Server" on page 12 and to "Displaying the LAN Configuration Summary (Agilent MXG)" on page 16.

If the instrument has been restored to the factory defaults from the LAN Setup menu the signal generator will revert to the values displayed in Table 1-1 on page 15. Refer to "Displaying the LAN Configuration Summary (Agilent MXG)" on page 16.

To reset the instrument LXI password to "agilent" and the LAN settings to their factory default values, press the following key sequence on the signal generator:

Utility > I/O Config > LAN Setup > Advanced Settings > Restore LAN Settings to Default Values > Restore LAN Settings to Default Values

NOTE There are no SCPI commands associated with this LXI password factory reset.

For more information, refer to the signal generator's Web Server Interface Help.

Table 1-1 LAN Configuration Summary Values

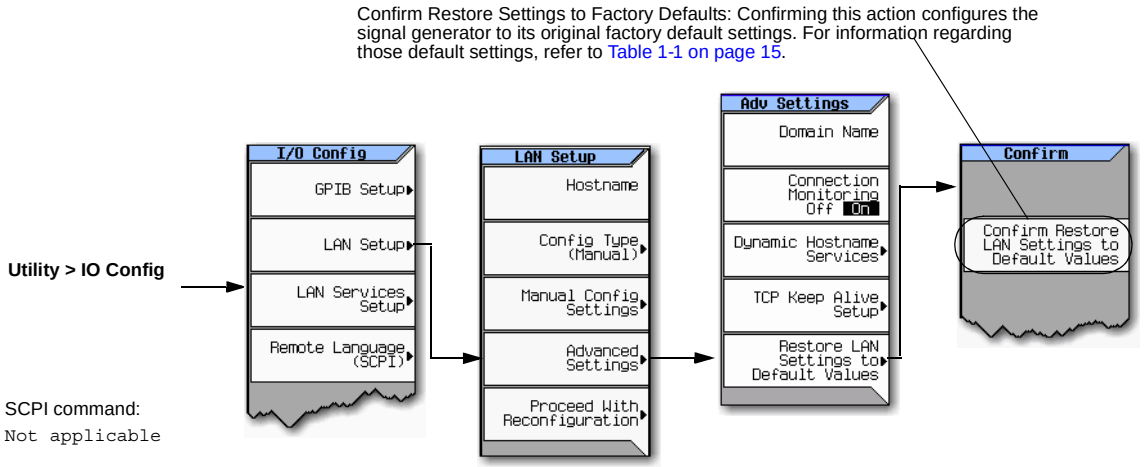
Parameter	Default
Signal Generator LAN Configuration Summary	
Hostname:	Agilent–<model number>–<last_5_chars_of_serial_number>
Config Type:	AUTO
IP Address:	127.0.0.1
Connection Monitoring:	On
Subnet:	255.255.255.0
DNS Server Override:	Off
Gateway:	0.0.0.0
Dynamic DNS Naming:	On
RFC NETBIOS Naming:	On
DNS Server:	0.0.0.0

Table 1-1 LAN Configuration Summary Values

Parameter	Default
TCP Keep Alive:	On
Domain Name:[a]	<empty>
TCP Keep Alive Timeout:	1800.0 sec
Signal Generator Web Server Interface	
Description:	Agilent <model_number>(<serial_number>)
SICL Interface Name[b]:	gpib0
Web Password:	agilent

a. The Domain Name defaults to a null field.
b. This information is part of the "Advanced Information about this Web-Enabled <signal generator model number>"

Displaying the LAN Configuration Summary (Agilent MXG)


Confirm Restore Settings to Factory Defaults: Confirming this action configures the signal generator to its original factory default settings. For information regarding those default settings, refer to Table 1-1 on page 15.
Utility > IO Config
I/O Config
GPIB Setup
LAN Setup
LAN Services Setup
Remote Language (SCPI)
LAN Setup
Hostname
Config Type (Manual)
Manual Config Settings
Advanced Settings
Proceed With Reconfiguration
Adv Settings
Domain Name
Connection Monitoring Off On
Dynamic Hostname Services
TCP Keep Alive Setup
Restore LAN Settings to Default Values
Confirm
Confirm Restore LAN Settings to Default Values
SCPI command:
Not applicable

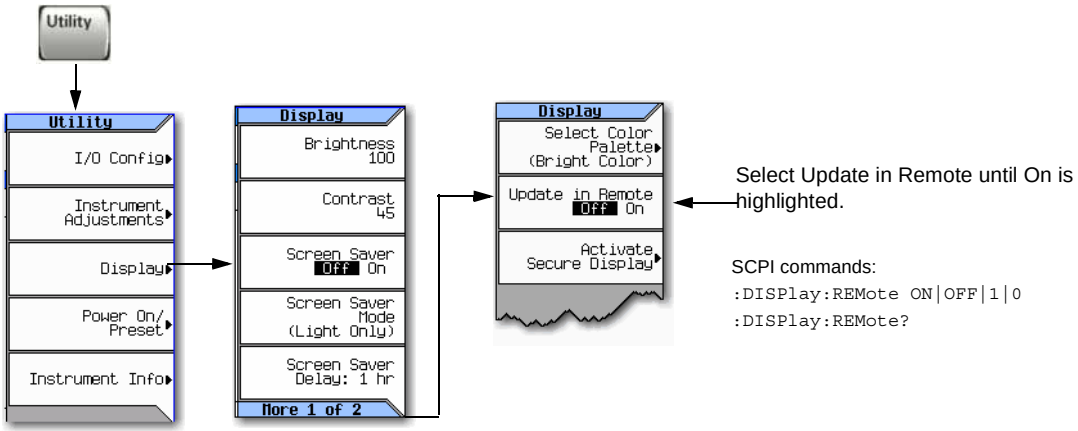
For details on each key, use the key help (described in the *User's Guide*).

Preferences

The following commonly-used manual command sections are included here:

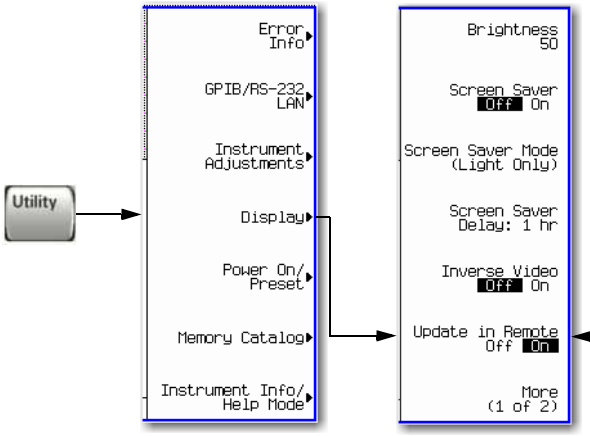
"Configuring the Display for Remote Command Setups (Agilent MXG)" on page 17
"Configuring the Display for Remote Command Setups (ESG/PSG/E8663B)" on page 18
"Getting Help (Agilent MXG)" on page 18
"Setting the Help Mode (ESG/PSG/E8663B)" on page 19
"Setting the Help Mode (ESG/PSG/E8663B)" on page 19

Configuring the Display for Remote Command Setups (Agilent MXG)



For details on each key, use the key help (described in *User's Guide*).

Configuring the Display for Remote Command Setups (ESG/PSG/E8663B)



SCPI commands:

```
:DISPlay:REMote ON|OFF|1|0
:DISPlay:REMote?
```

Select Update in Remote until On is highlighted.

Using the Update in Remote softkey updates the display but *not* the softkeys on each SCPI command.

In general, the softkeys are not updated until the SCPI command `SYST:DISP:GTL` is sent.

For details on each key, use the *Key and Data Field Reference*. For additional SCPI command information, refer to the SCPI Command Reference.

Getting Help (Agilent MXG)

When you press **Help**:

Help displays for the next key you press. Use the cursor keys, Page Up, Page Down, and the RPG knob to scroll the help text. Then press Cancel to close the help window or press any other key to close the help window and execute that key.

For details on each key, use the key help (described in *User's Guide*).

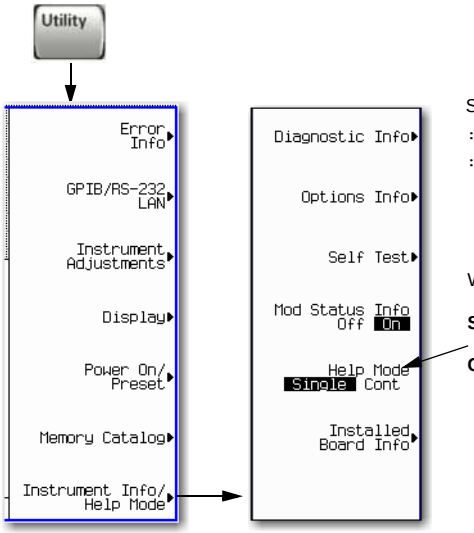
Getting Help (ESG/PSG/E8663B)

When you press **Help**:

Help displays for the next key you press or you see help for the next key or for every key, depending on the Help mode.

For details on each key, use the key help (described in *User's Guide*).

Setting the Help Mode (ESG/PSG/E8663B)



SCPI commands:

```
:SYSTem:HELP:MODE SINGle|CONTinuous
:SYSTem:HELP:MODE?
```

When you press **Help**:

Single: Help displays only for the next key you press.

Cont: Help displays for each key you press *and* that key's function activates. To turn off the function, press **Help**.

For details on each key, use the *Key and Data Field Reference*. For additional SCPI command information, refer to the SCPI Command Reference.

Troubleshooting

In each section of this document, there is information that is related to troubleshooting that topic, if applicable. Refer to those corresponding sections in this document as well as to the *User's Guide*, before using the diagnostics mode referred to in the *Service Guide* and in the caution below.

CAUTION All MXGs have a fail-safe and diagnostic mode that should *only* be used if all other troubleshooting mentioned in this document has been attempted and failed. If the diagnostic mode is determined to be needed, refer to the *Service Guide*.

The fail-safe and diagnostic mode can be enabled on the N5161A/62A signal generators by pressing a combination of the front panel hardkeys: LAN Reset and the power switch during boot-up. *This fail-safe and diagnostic mode should rarely be used as the instrument's data could be permanently damaged.* Refer to the *Service Guide*.

- If the diagnostic mode fails to function, refer to the *Service Guide*.

Error Messages

If an error condition occurs in the signal generator, it is reported to both the SCPI (remote interface) error queue and the front panel display error queue. These two queues are viewed and managed separately; for information on the front panel display error queue, refer to the *User's Guide*.

NOTE For additional general information on troubleshooting problems with your connections, refer to the Help in the Agilent IO Libraries and documentation.

When accessing error messages using the SCPI (remote interface) error queue, the error numbers and the <error_description> portions of the error query response are displayed on the host terminal.

Characteristic	SCPI Remote Interface Error Queue
Capacity (#errors)	30
Overflow Handling	Linear, first-in/first-out. Replaces newest error with: `-350, Queue overflow`
Viewing Entries[a]	Use SCPI query `SYSTem:ERRor[:NEXT]?`
Clearing the Queue[b]	Power up Send a `*CLS` command Read last item in the queue
Unresolved Errors[c]	Re-reported after queue is cleared.
No Errors	When the queue is empty (every error in the queue has been read, or the queue is cleared), the following message appears in the queue: `+0, "No error"`

a. On the Agilent MXG, using this SCPI command to read out the error messages clears the display of the ERR annunciator and the error message at the bottom of the screen.
b. On the Agilent MXG, executing the SCPI command *CLS clears the display of the ERR annunciator and the error message at the bottom of the screen.
c. Errors that still exist after clearing the error queue. For example, unlock.

Error Message File

A complete list of error messages is provided in the file *errormessages.pdf*, on the CD-ROM supplied with your instrument. In the error message list, an explanation is generally included with each error to further clarify its meaning. The error messages are listed numerically. In cases where there are multiple listings for the same error number, the messages are in alphabetical order.

Error Message Types

Events generate only one type of error. For example, an event that generates a query error will not generate a device-specific, execution, or command error.

Query Errors (–499 to –400) indicate that the instrument's output queue control has detected a problem with the message exchange protocol described in IEEE 488.2, Chapter 6. Errors in this class set the query error bit (bit 2) in the event status register (IEEE 488.2, section 11.5.1). These errors correspond to message exchange protocol errors described in IEEE 488.2, 6.5. In this case:

- Either an attempt is being made to read data from the output queue when no output is either present or pending, or
- data in the output queue has been lost.

Device Specific Errors (–399 to –300, 201 to 703, and 800 to 810) indicate that a device operation did not properly complete, possibly due to an abnormal hardware or firmware condition. These codes are also used for self-test response errors. Errors in this class set the device-specific error bit (bit 3) in the event status register (IEEE 488.2, section 11.5.1).

The <error_message> string for a *positive* error is not defined by SCPI. A positive error indicates that the instrument detected an error within the GPIB system, within the instrument's firmware or hardware, during the transfer of block data, or during calibration.

Execution Errors (–299 to –200) indicate that an error has been detected by the instrument's execution control block. Errors in this class set the execution error bit (bit 4) in the event status register (IEEE 488.2, section 11.5.1). In this case:

- Either a <PROGRAM DATA> element following a header was evaluated by the device as outside of its legal input range or is otherwise inconsistent with the device's capabilities, or
- a valid program message could not be properly executed due to some device condition.

Execution errors are reported *after* rounding and expression evaluation operations are completed. Rounding a numeric data element, for example, is not reported as an execution error.

Command Errors (–199 to –100) indicate that the instrument's parser detected an IEEE 488.2 syntax error. Errors in this class set the command error bit (bit 5) in the event status register (IEEE 488.2, section 11.5.1). In this case:

- Either an IEEE 488.2 syntax error has been detected by the parser (a control-to-device message was received that is in violation of the IEEE 488.2 standard. Possible violations include a data element that violates device listening formats or whose type is unacceptable to the device.), or
- an unrecognized header was received. These include incorrect device-specific headers and incorrect or unimplemented IEEE 488.2 common commands.

2 Using IO Interfaces

NOTE For the N5161A/62A the softkey menus and features mentioned in this chapter are only available through the Web–Enabled MXG or via SCPI commands. Refer to "Using the Web Browser" on page 11 and to the *SCPI Command Reference*.

Using the programming examples with GPIB, LAN, RS–232, and USB interfaces:

- Using GPIB on page 23
- Using LAN on page 29
- Using RS–232 (ESG, PSG, and E8663B Only) on page 58
- Using USB (Agilent MXG) on page 66

Using GPIB

GPIB enables instruments to be connected together and controlled by a computer. GPIB and its associated interface operations are defined in the ANSI/IEEE Standard 488.1–1987 and ANSI/IEEE Standard 488.2–1992. See the IEEE website, *http://www.ieee.org*, for details on these standards.

The following sections contain information for installing a GPIB interface card or NI–GPIB interface card for your PC or UNIX–based system.

- "Installing the GPIB Interface" on page 23
- "Set Up the GPIB Interface" on page 25
- "Verify GPIB Functionality" on page 26

Installing the GPIB Interface

NOTE You can also connect GPIB instruments to a PC USB port using the Agilent 82357A USB/GPIB Interface Converter, which eliminates the need for a GPIB card. For more information, refer to the table on page 24 or go to *http://www.agilent.com/find/gpib*.

A GPIB interface card can be installed in a computer. Two common GPIB interface cards are the Agilent GPIB interface card and the National Instruments (NI) PCI–GPIB card. Follow the interface card instructions for installing and configuring the card. The following table provide lists on some of the available interface cards. Also, see the Agilent website, http://*www.agilent.com* for details on GPIB interface cards.

Interface Type	Operating System	IO Library	Languages	Backplane/ BUS	Max IO (kB/sec)	Buffering
Agilent USB/GPIB Interface Converter for PC-Based Systems						
Agilent 82357A Converter	Windows[a] 98(SE)/ME/ 2000®/XP	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows, NI Labview	USB 2.0 (1.1 compatible)	850	Built–in
Agilent GPIB Interface Card for PC-Based Systems						
Agilent 82341C for ISA bus computers	Windows[b] 95/98/NT /2000®	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	ISA/EISA, 16 bit	750	Built–in
Agilent 82341D Plug&Play for PC	Windows 95	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	ISA/EISA, 16 bit	750	Built–in
Agilent 82350A for PCI bus computers	Windows 95/98/NT /2000	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	PCI 32 bit	750	Built–in
Agilent 82350B for PCI bus computers	Windows 98(SE)/ME/2000 /XP	VISA / SICL	C/C++, Visual Basic, Agilent VEE, HP Basic for Windows	PCI 32 bit	> 900	Built–in
NI-GPIB Interface Card for PC-Based Systems						
National Instruments PCI–GPIB	Windows 95/98/2000/ ME/NT	VISA NI–488.2 [c]	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MBps	Built–in
National Instruments PCI–GPIB+	Windows NT	VISA NI–488.2	C/C++, Visual BASIC, LabView	PCI 32 bit	1.5 MBps	Built–in
Agilent-GPIB Interface Card for HP-UX Workstations						
Agilent E2071C	HP–UX 9.x, HP–UX 10.01	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP–UX	EISA	750	Built–in
Agilent E2071D	HP–UX 10.20	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP–UX	EISA	750	Built–in

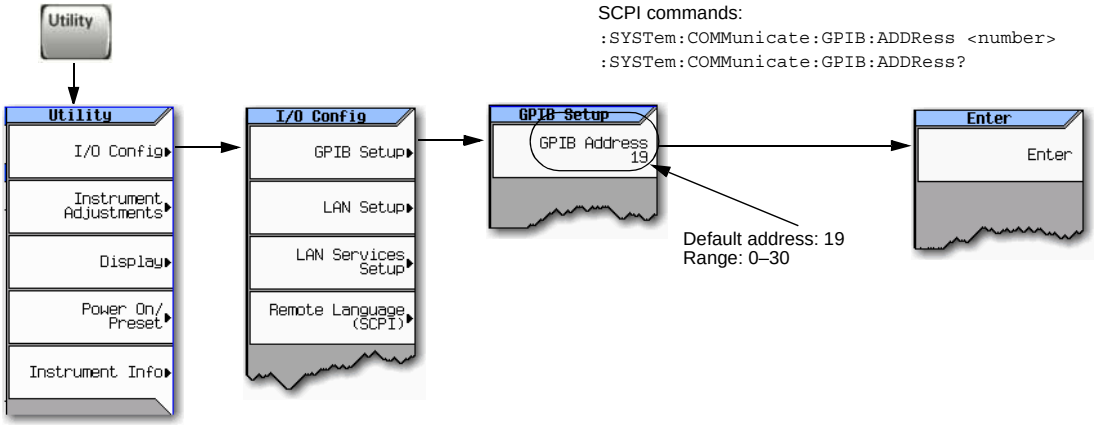
Interface Type	Operating System	IO Library	Languages	Backplane/ BUS	Max IO (kB/sec)	Buffering
Agilent USB/GPIB Interface Converter for PC-Based Systems						
Agilent E2078A	HP–UX 10.20	VISA/SICL	ANSI C, Agilent VEE, Agilent BASIC, HP–UX	PCI	750	Built–in

a. Windows 95, 98(SE), NT, 2000, and XP are registered trademarks of Microsoft Corporation.
b. Windows 98 and ME are registered trademarks of Microsoft Corporation.
c. NI–488.2 is a trademark of National Instruments Corporation.

Set Up the GPIB Interface

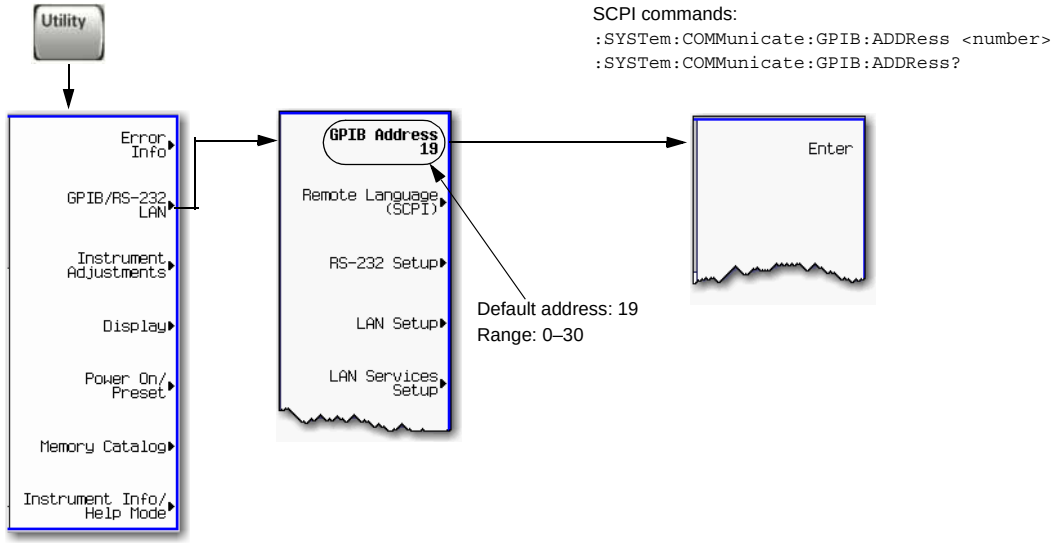
For the Agilent MXG refer to the Figure 2-1 and for the ESG, PSG, and E8663B, Figure 2-2 on page 26.

Figure 2-1 Setting the GPIB Address on the Agilent MXG



For details on each key, use the key help. Refer to "Getting Help (Agilent MXG)" on page 18 and the *User's Guide*. For additional SCPI command information, refer to the *SCPI Command Reference*.

Figure 2-2 Setting the GPIB Address on the ESG/PSG/E8663B



For details on each key, use the *Key and Data Field Reference*. For additional SCPI command information, refer to the *SCPI Command Reference*.

Connect a GPIB interface cable between the signal generator and the computer. (The following table lists cable part numbers.)

Model	10833A	10833B	10833C	10833D	10833F	10833G
Length	1 meter	2 meters	4 meters	.5 meter	6 meters	8 meters

Verify GPIB Functionality

To verify GPIB functionality, use the VISA Assistant, available with the Agilent IO Library or the Getting Started Wizard available with the National Instrument IO Library. These utility programs enable you to communicate with the signal generator and verify its operation over GPIB. For information and instructions on running these programs, refer to the Help menu available in each utility.

If You Have Problems

1. Verify that the signal generator's address matches the address declared in the program (example programs in Chapter 2 use address 19).
2. Remove all other instruments connected via GPIB and rerun the program.
3. Verify that the GPIB card's name or id number matches the GPIB name or id number configured for your PC.

GPIB Interface Terms

An instrument that is part of a GPIB network is categorized as a listener, talker, or controller, depending on its current function in the network.

listener	A listener is a device capable of receiving data or commands from other instruments. Several instruments in the GPIB network can be listeners simultaneously.
talker	A talker is a device capable of transmitting data. To avoid confusion, a GPIB system allows only one device at a time to be an active talker.
controller	A controller, typically a computer, can specify the talker and listeners (including itself) for an information transfer. Only one device at a time can be an active controller.

GPIB Programming Interface Examples

NOTE The portions of the programming examples discussed in this section are taken from the full text of these programs that can be found in Chapter 3, "Programming Examples."

- "Interface Check using HP Basic and GPIB" on page 27
- "Interface Check Using NI–488.2 and C++" on page 28

Before Using the GPIB Examples

If the Agilent GPIB interface card is used, the Agilent VISA library should be installed along with Agilent SICL. If the National Instruments PCI–GPIB interface card is used, the NI–VISA library along with the NI–488.2 library should be installed. Refer to "Selecting IO Libraries for GPIB" on page 9 and the documentation for your GPIB interface card for details.

HP Basic addresses the signal generator at 719. The GPIB card is addressed at 7 and the signal generator at 19. The GPIB address designator for other libraries is typically GPIB0 or GPIB1.

The following sections contain HP Basic and C++ lines of programming removed from the programming interface examples in Chapter 3, "Programming Examples." these portions of programming demonstrate the important features to consider when developing programming for use with the GPIB interface.

Interface Check using HP Basic and GPIB

This portion of the example program "Interface Check using HP Basic and GPIB" on page 27, causes the signal generator to perform an instrument reset. The SCPI command `*RST` places the signal generator into a pre–defined state and the remote annunciator (`R`) appears on the front panel display.

The following program example is available on the signal generator Documentation CD–ROM as basicex1.txt. For the full text of this program, refer to "Interface Check using HP Basic and GPIB" on page 78 or to the signal generator's documentation CD–ROM.

```
160   Sig_gen=719    ! Declares a variable to hold the signal generator's address
```

```
170   LOCAL Sig_gen  ! Places the signal generator into Local mode
180   CLEAR Sig_gen  ! Clears any pending data I/O and resets the parser
190   REMOTE 719     ! Puts the signal generator into remote mode
200   CLEAR SCREEN   ! Clears the controllers display
210   REMOTE 719
220   OUTPUT Sig_gen;"*RST"  ! Places the signal generator into a defined state
```

Interface Check Using NI–488.2 and C++

This portion of the example program "Interface Check Using NI–488.2 and C++" on page 28, uses the NI–488.2 library to verify that the GPIB connections and interface are functional.

The following program example is available on the signal generator Documentation CD–ROM as `niex1.cpp`. For the full text of this program, refer to "Interface Check Using NI-488.2 and C++" on page 79 or to the signal generator's documentation CD–ROM.

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;

int GPIB0=   0;         // Board handle
Addr4882_t Address[31]; // Declares an array of type Addr4882_t

int main(void)

{
     int sig;                          // Declares a device descriptor variable
     sig = ibdev(0, 19, 0, 13, 1, 0); // Aquires a device descriptor
     ibclr(sig);                       // Sends device clear message to signal generator
     ibwrt(sig, "*RST", 4);            // Places the signal generator into a defined state
```

Using LAN

The Agilent MXG is capable of 100Base–T LAN communication. The ESG, PSG, and E8663B are designed to connect with a 10Base–T LAN. Where auto–negotiation is present, the ESG, PSG, and E8663B can connect to a 100Base–T LAN, but communicate at 10Base–T speeds. For more information refer to *http://www.ieee.org*.

The signal generator can be remotely programmed via a 100Base–T LAN interface or 10Base–T LAN interface and LAN–connected computer using one of several LAN interface protocols. The LAN allows instruments to be connected together and controlled by a LAN–based computer. LAN and its associated interface operations are defined in the IEEE 802.2 standard. For more information refer to http://www.ieee.org.

NOTE For more information on configuring your signal generator for LAN, refer to the *User's Guide* for your signal generator. Also, for the Agilent MXG, refer to *www.agilent.com* and search on the *FAQs: Hardware Configurations and Installation*.

The signal generator supports the following LAN interface protocols:

- VXI–11 (See page 39)
- Sockets LAN (See page 40)
- Telephone Network (TELNET) (See page 41)
- File Transfer Protocol (FTP) (See page 45)
- LXI (See page 47)

VXI–11 and sockets LAN are used for general programming using the LAN interface, TELNET is used for interactive, one command at a time instrument control, and FTP is for file transfer. LXI is used to communicate with multiple instruments via LAN events using precision time protocols.

For more information on the LXI standards, refer to *www.agilent.com/find/*.

NOTE For more information on configuring the signal generator to communicate over the LAN, refer to "Using VXI–11" on page 39.

The following sections contain information on selecting and connecting IO libraries and LAN interface hardware that are required to remotely program the signal generator via LAN to a LAN–based computer and combining those choices with one of several possible LAN interface protocols.

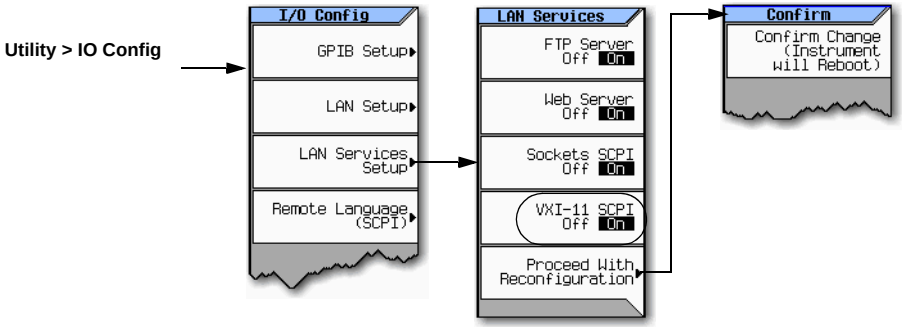
- "Setting Up the LAN Interface" on page 30
- "Verifying LAN Functionality" on page 34

Setting Up the LAN Interface

For LAN operation, the signal generator must be connected to the LAN, and an IP address must be assigned to the signal generator either manually or by using DHCP client service. Your system administrator can tell you which method to use. (Most modern LAN networks use DHCP.)

NOTE Verify that the signal generator is connected to the LAN using a 100Base–T LAN or 10Base–T LAN cable.

Configuring the VXI–11 for LAN (Agilent MXG)



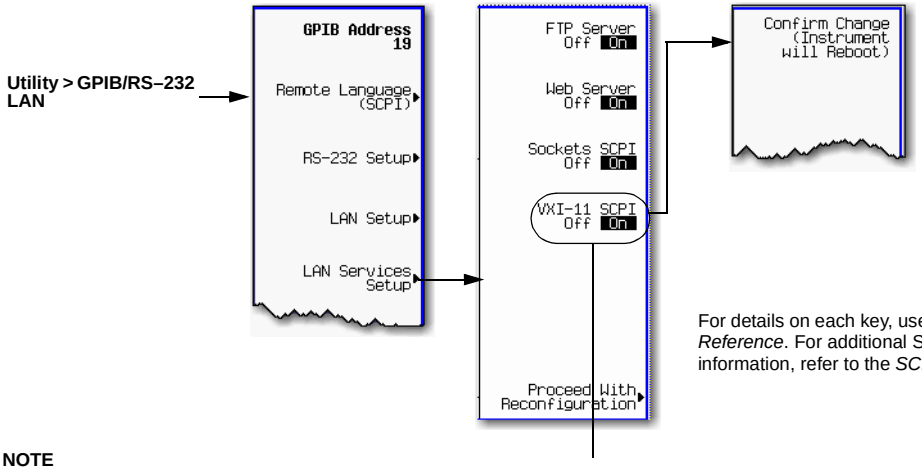
NOTE

To communicate with the signal generator over the LAN, you must enable the VXI–11 SCPI service. Select VXI–11 until On is highlighted. (Default condition is On.)

For optimum performance, use a 100Base–T LAN cable to connect the signal generator to the LAN.

For details on each key, use the key help. For information describing the key help, refer to "Getting Help (Agilent MXG)" on page 18 and the *User's Guide*. For additional SCPI command information, refer to the *SCPI Command Reference*.

Configuring the VXI–11 for LAN (ESG/PSG/E8663B)



For details on each key, use the *Key and Data Field Reference*. For additional SCPI command information, refer to the *SCPI Command Reference*.

NOTE
To communicate with the signal generator over the LAN, you must enable the VXI–11 SCPI service. Select VXI–11 until On is highlighted. (Default condition is On.)

Use a 10Base–T LAN cable to connect the signal generator to the LAN.Where auto–negotiation is present, the ESG, PSG, or E8663B can connect to 100Base–T LAN, but will communicate at 10Base–T speeds. For more information refer to *http://www.ieee.org*.

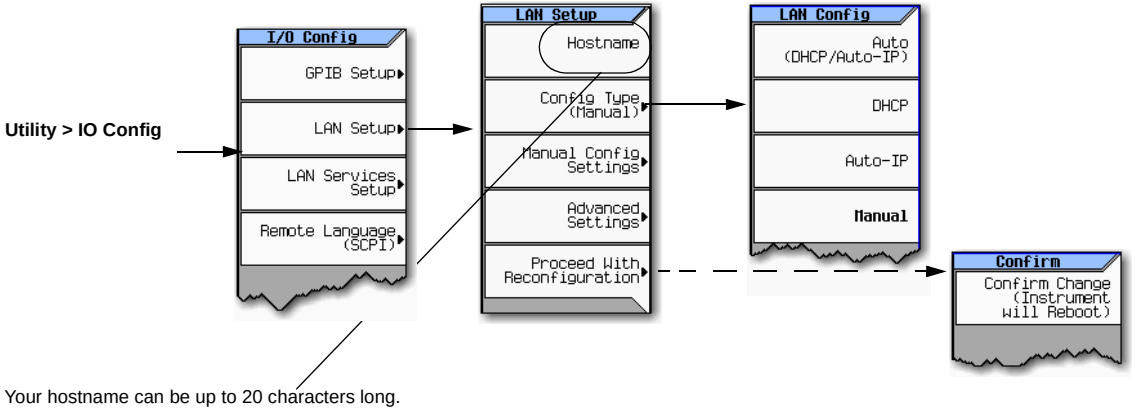
Manual Configuration

The **Hostname** softkey is only available when **LAN Config Manual DHCP** is set to **Manual**.

To remotely access the signal generator from a different LAN subnet, you must also enter the subnet mask and default gateway. See your system administrator for more information.

For more information on the manual configuration, refer to "Manually Configuring the Agilent MXG LAN" on page 32 or to "Manually Configuring the ESG/PSG/E8663B LAN" on page 32.

Manually Configuring the Agilent MXG LAN



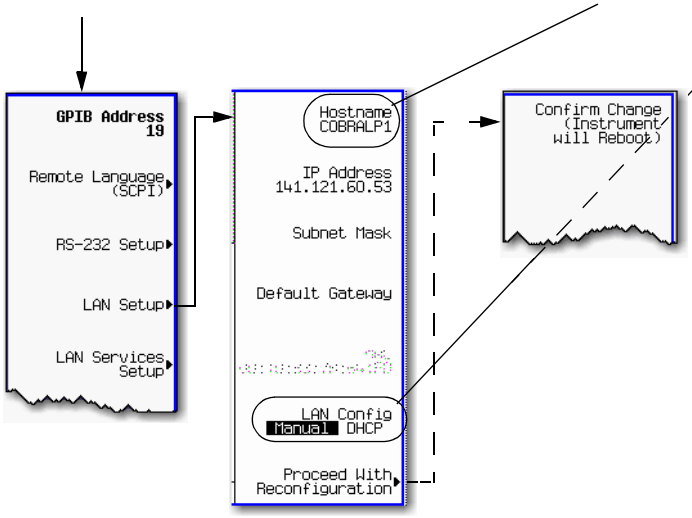

```
SCPI commands:
:SYSTem:COMMunicate:LAN:CONFig MANual
:SYSTem:COMMunicate:LAN:CONFig?
```

For details on each key, use the key help (described in *User's Guide*). For additional SCPI command information, refer to the *SCPI Command Reference*.

Manually Configuring the ESG/PSG/E8663B LAN

Utility > IO Config

The Hostname softkey is available only when LAN Config Manual DHCP is set to Manual. Your hostname can be up to 20 characters long.




```
SCPI commands:
:SYSTem:COMMunicate:LAN:CONFig MANual
:SYSTem:COMMunicate:LAN:CONFig?
```

For details on each key, use the *Key and Data Field Reference*. For additional SCPI command information, refer to the *SCPI Command Reference*.

DHCP Configuration

If the DHCP server uses dynamic DNS to link the hostname with the assigned IP address, the hostname may be used in place of the IP address. Otherwise, the hostname is not usable.

For more information on the DHCP configuration, refer to "Configuring the DHCP LAN (Agilent MXG)" on page 33 or "Configuring the DHCP LAN (ESG/PSG/E8663B)" on page 34.

AUTO (DHCP/Auto–IP) Configuration (Agilent MXG)

DHCP and Auto–IP are used together to make automatic (AUTO) mode for IP configuration. Automatic mode attempts DHCP first and then if that fails Auto–IP is used to detect a private network. If neither is found, Manual is the final choice.

If the DHCP server uses dynamic DNS to link the hostname with the assigned IP address, the hostname may be used in place of the IP address. Otherwise, the hostname is not usable.

Auto–IP provides automatic TCP/IP set–up for instruments on any manually configured networks.

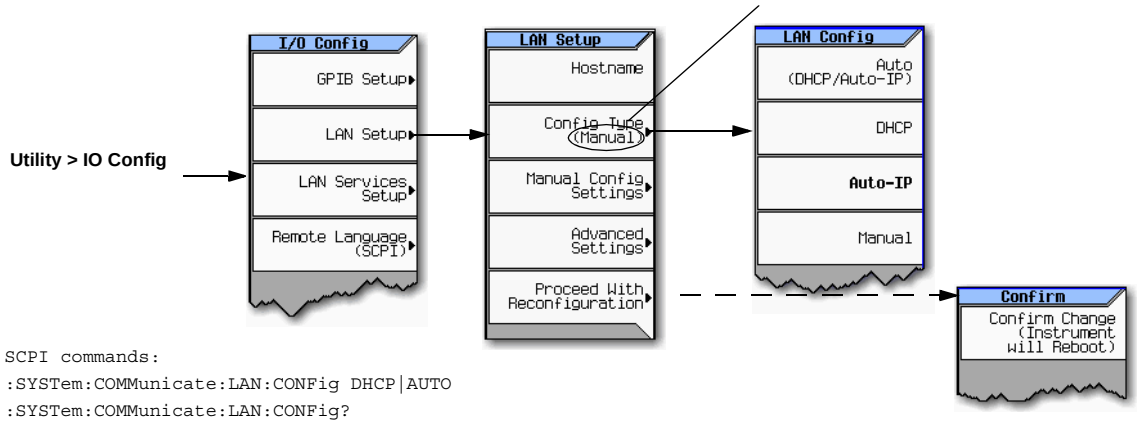
For more information on the AUTO (DHCP/Auto–IP) configuration, refer to "Configuring the DHCP LAN (Agilent MXG)" on page 33.

Configuring the DHCP LAN (Agilent MXG)

AUTO (DHCP/Auto–IP): Request a new IP address in the following sequence: 1) from the DHCP (server–based LAN), 2) Auto–IP (private network without a network administrator) or if neither is available, 3) Manual setting is selected.

DHCP: Request a new IP address from the DHCP server each power cycle.

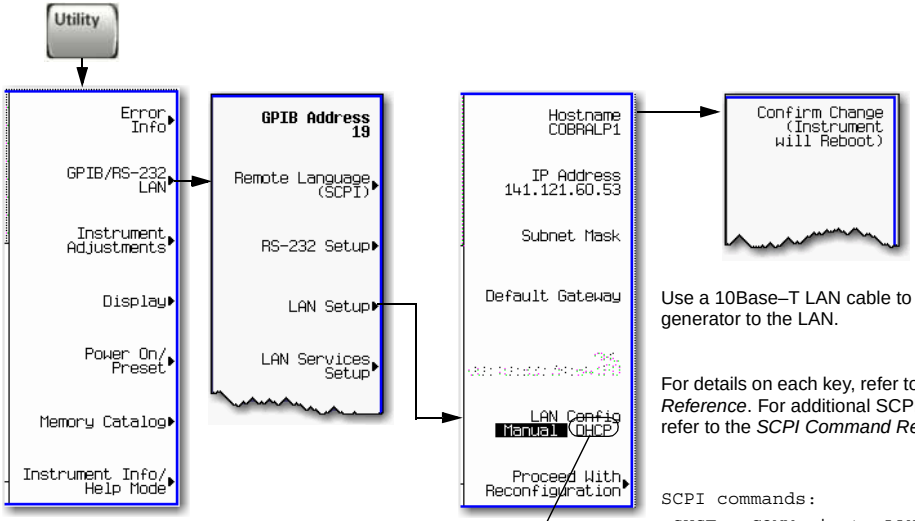
Confirming this action configures the signal generator as a DHCP client. In DHCP mode, the signal generator will request a new IP address from the DHCP server upon rebooting to determine the assigned IP address.




```
SCPI commands:
:SYSTem:COMMunicate:LAN:CONFig DHCP|AUTO
:SYSTem:COMMunicate:LAN:CONFig?
```

For details on each key, use the key help (described in *User's Guide*). For additional SCPI command information, refer to the *SCPI Command Reference*.

Configuring the DHCP LAN (ESG/PSG/E8663B)



DHCP: Request a new IP address from the DHCP server each power cycle. Confirming this action configures the signal generator as a DHCP client. In DHCP mode, the signal generator will request a new IP address from the DHCP server upon rebooting to determine the assigned IP address.

NOTE

Use a 10Base–T LAN cable to connect the signal generator to the LAN.

For details on each key, refer to the *Key and Data Field Reference*. For additional SCPI command information, refer to the *SCPI Command Reference*.

SCPI commands:

```
:SYSTem:COMMunicate:LAN:CONFig DHCP
:SYSTem:COMMunicate:LAN:CONFig?
```

Setting up Private LAN

You can connect the Agilent MXG, ESG, PSG or E8663B directly to a PC using a crossover cable. To do this, you should either choose to set IP addresses of the PC and signal generator to differ only in the last digit (example: PC's IP: 1.1.1.1 and Signal generator's IP: 1.1.1.2); or you can use the DHCP feature or Auto–IP feature if your PC supports them. For more information go to *www.agilent.com*, and search on the *Connectivity Guide* (E2094–90009) or use the Agilent Connection Expert's Help to see the *Connection Guide*.

Verifying LAN Functionality

Verify the communications link between the computer and the signal generator remote file server using the ping utility. Compare your ping response to those described in "LAN Ping Responses" on page 36.

NOTE For additional information on troubleshooting your LAN connection, refer to "If You Have Problems" on page 35 and to the Help in the Agilent IO Libraries and documentation for LAN connections and problems.

From a UNIX® workstation, type (UNIX is a registered trademark of the Open Group):

```
ping <hostname or IP address> 64 10
```

where `<hostname or IP address>` is your instrument's name or IP address, 64 is the packet size, and 10 is the number of packets transmitted. Type `man ping` at the UNIX prompt for details on the ping command.

From the MS–DOS® Command Prompt or Windows environment, type:

```
ping -n 10 <hostname or IP address>
```

where `<hostname or IP address>` is your instrument's name or IP address and 10 is the number of echo requests. Type `ping` at the command prompt for details on the ping command.

NOTE In DHCP mode, if the DHCP server uses dynamic DNS to link the hostname with the assigned IP address, the hostname may be used in place of the IP address. Otherwise, the hostname is not usable and you must use the IP address to communicate with the signal generator over the LAN.

If You Have Problems

If you are experiencing problems with the LAN connection on the signal generator, verify the rear panel LAN connector green LED is on.

For additional information on troubleshooting your LAN connection, refer to the Help in the Agilent IO Libraries and documentation for LAN connections and problems.

NOTE The N5161A/62A signal generators have a **LAN Reset** hardkey on the front panel, that can be pressed to reset the instrument to the factory default conditions. Pressing the **LAN Reset** hardkey is equivalent to sending the following SCPI commands:

```
:SYSTem:COMMunicate:LAN:DEFaults
:SYSTem:COMMunicate:LAN:RESTart
```

When the hardkey is pressed the hostname defaults to the factory default of: a–<*model_number*>–<(*serial_number*)>. Example: `a-n5162a-00283`

MS–DOS, and Visual Basic are registered trademarks of Microsoft.

LAN Ping Responses

Normal Response for UNIX	A normal response to the ping command will be a total of 9 or 10 packets received with a minimal average round–trip time. The minimal average will be different from network to network. LAN traffic will cause the round–trip time to vary widely.
Normal Response for DOS or Windows	A normal response to the ping command will be a total of 9 or 10 packets received if 10 echo requests were specified.
Error Messages	If error messages appear, then check the command syntax before continuing with troubleshooting. If the syntax is correct, resolve the error messages using your network documentation or by consulting your network administrator. If an unknown host error message appears, try using the IP address instead of the hostname. Also, verify that the host name and IP address for the signal generator have been registered by your IT administrator. Check that the hostname and IP address are correctly entered in the node names database. To do this, enter the `nslookup <hostname>` command from the command prompt.
No Response	If there is no response from a ping, no packets were received. Check that the typed address or hostname matches the IP address or hostname assigned to the signal generator in the System **LAN Setup** menu. For more information, refer to "Configuring the DHCP LAN (Agilent MXG)" on page 33 or "Configuring the DHCP LAN (ESG/PSG/E8663B)" on page 34. Ping each node along the route between your workstation and the signal generator, starting with your workstation. If a node doesn't respond, contact your IT administrator. If the signal generator still does not respond to ping, you should suspect a hardware problem. • Check the signal generator LAN connector lights • Verify the hostname is not being used with DHCP addressing
Intermittent Response	If you received 1 to 8 packets back, there maybe a problem with the network. In networks with switches and bridges, the first few pings may be lost until these devices 'learn' the location of hosts. Also, because the number of packets received depends on your network traffic and integrity, the number might be different for your network. Problems of this nature are best resolved by your IT department.

Using Interactive IO

Use the VISA Assistant utility available in the Agilent IO Libraries Suite to verify instrument communication over the LAN interface. Refer to the section on the "IO Libraries and Programming Languages" on page 5 for more information.

The Agilent IO Libraries Suite is supported on all platforms except Windows NT. If you are using Windows NT, refer to the section below on using the VISA Assistant to verify LAN communication. See the section on "Windows NT and Agilent IO Libraries M (and Earlier)" on page 7 for more information.

NOTE The following sections are specific to Agilent IO Libraries versions M and earlier and apply only to the Windows NT platform.

VISA Configuration

Use the Agilent IO Libraries Suite Version 15.0, to perform the following steps to use the Connection Expert and VISA to manually configure an interface.

1. Run the Connection Expert program.
2. On the tool bar select the **Add Interface** button.
3. Click on **LAN Interface** in the **Available interface types** text box.
4. Click the **ADD** button.
5. Verify that the **Auto (automatically detect protocol)** bubble is checked. Click **O.K.** to use the default settings.
6. Click on **LAN(TCPIP0)** in the **Instrument I/O on this PC** text box.
7. On the tool bar select the **Add Instrument** button.
8. Click the **Add Address** button in the **Add LAN Instruments** window.
9. Enter the *hostname* of the instrument or select the **Use IP Address** check box and enter the IP address.
10. Click the **OK** button in this form.

Using VISA Assistant

Use the VISA Assistant, available with the Agilent IO Library versions M and earlier, to communicate with the signal generator over the LAN interface. However, you must manually configure the VISA LAN client. Refer to the Help menu for instructions on configuring and running the VISA Assistant program.

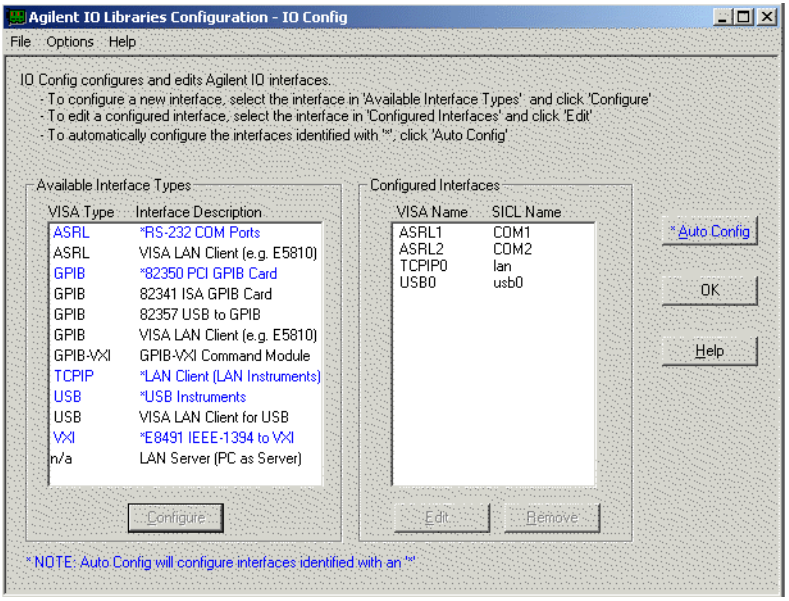
1. Run the IO Config program.
2. Click on TCPIP0 in the Available Interface Types text box.
3. Click the **Configure** button. Then Click **OK** to use the default settings.
4. Click on **TCPIP0** in the Configured Interfaces text box.
5. Click **Edit**...
6. Click the **Edit VISA Config**... button.
7. Click the **Add device** button.
8. Enter the TCPIP address of the signal generator. Leave the Device text box empty.
9. Click the **OK** button in this form and all subsequent forms to exit the IO Config program.

If You Have Problems

1. Verify the signal generator's IP address is valid and that no other instrument is using the IP address.
2. Switch between manual LAN configuration and DHCP using the front panel **LAN Config** softkey and run the ping program using the different IP addresses.

NOTE For Agilent IO Libraries versions M and earlier, you must manually configure the VISA LAN client in the IO Config program if you want to use the VISA Assistant to verify LAN configuration. Refer to the IO Libraries Installation Guide for information on configuring IO interfaces. The IO Config program interface is shown in Figure 2-4 on page 40.

Figure 2-3 IO Config Form (Windows NT)


Agilent IO Libraries Configuration - IO Config
File Options Help
IO Config configures and edits Agilent IO interfaces.
- To configure a new interface, select the interface in 'Available Interface Types' and click 'Configure'
- To edit a configured interface, select the interface in 'Configured Interfaces' and click 'Edit'
- To automatically configure the interfaces identified with '*', click 'Auto Config'
Available Interface Types
VISA Type Interface Description
ASRL *RS-232 COM Ports
ASRL VISA LAN Client (e.g. E5810)
GPIB *82350 PCI GPIB Card
GPIB 82341 ISA GPIB Card
GPIB 82357 USB to GPIB
GPIB VISA LAN Client (e.g. E5810)
GPIB-VXI GPIB-VXI Command Module
TCPIP *LAN Client (LAN Instruments)
USB *USB Instruments
USB VISA LAN Client for USB
VXI *E8491 IEEE-1394 to VXI
n/a LAN Server (PC as Server)
Configure...
Configured Interfaces
VISA Name SICL Name
ASRL1 COM1
ASRL2 COM2
TCPIP0 lan
USB0 usb0
Edit...
Remove
* Auto Config
OK
Help
* NOTE: Auto Config will configure interfaces identified with an '*'

Check to see that the Default Protocol is set to Automatic.

1. Run the IO Config program.
2. Click on TCPIP in the Configured Interfaces text box. If there is no TCPIP0 in the box, follow the steps shown in the section "Using VISA Assistant" on page 37.
3. Click the **Edit** button.
4. Click the radio button for AUTO (automatically detect protocol).
5. Click **OK**, **OK** to end the IO Config program.

Using VXI–11

The signal generator supports the LAN interface protocol described in the VXI–11 standard. VXI–11 is an instrument control protocol based on Open Network Computing/Remote Procedure Call (ONC/RPC) interfaces running over TCP/IP. It is intended to provide GBIB capabilities such as SRQ (Service Request), status byte reading, and DCAS (Device Clear State) over a LAN interface. This protocol is a good choice for migrating from GPIB to LAN as it has full Agilent VISA/SICL support.

NOTE It is recommended that the VXI–11 protocol be used for instrument communication over the LAN interface.

Configuring for VXI–11

The Agilent IO library has a program, IO Config, that is used to setup the computer/signal generator interface for the VXI–11 protocol. Download the latest version of the Agilent IO library from the Agilent website. Refer to the Agilent IO library user manual, documentation, and Help menu for information on running the IO Config program and configuring the VXI–11 interface.

Use the IO Config program to configure the LAN client. Once the computer is configured for a LAN client, you can use the VXI–11 protocol and the VISA library to send SCPI commands to the signal generator over the LAN interface. Example programs for this protocol are included in “LAN Programming Interface Examples” on page 112 of this programming guide.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI–11 SCPI service. For more information, refer to “Configuring the DHCP LAN (Agilent MXG)” on page 33 and “Configuring the DHCP LAN (ESG/PSG/E8663B)” on page 34.

If you are using the Windows NT platform, refer to “Windows NT and Agilent IO Libraries M (and Earlier)” on page 7 for information on using Agilent IO Libraries versions M or earlier to configure the interface.

For Agilent IO library version J.01.0100, the “Identify devices at run–time” check box must be unchecked. Refer to Figure 2-4.

Figure 2-4 Show Devices Form (Agilent IO Library version J.01.0100)

Using Sockets LAN

NOTE Users with Windows XP operating systems and newer can use this section to better understand how to use the signal generator with port settings. For more information, refer to the help software of the IO libraries being used.

Sockets LAN is a method used to communicate with the signal generator over the LAN interface using the Transmission Control Protocol/Internet Protocol (TCP/IP). A socket is a fundamental technology used for computer networking and allows applications to communicate using standard mechanisms built into network hardware and operating systems. The method accesses a port on the signal generator from which bidirectional communication with a network computer can be established.

Sockets LAN can be described as an internet address that combines Internet Protocol (IP) with a device port number and represents a single connection between two pieces of software. The socket can be accessed using code libraries packaged with the computer operating system. Two common versions of socket libraries are the Berkeley Sockets Library for UNIX systems and Winsock for Microsoft operating systems.

Your signal generator implements a sockets Applications Programming Interface (API) that is compatible with Berkeley socket for UNIX systems, and Winsock for Microsoft systems. The signal generator is also compatible with other standard sockets APIs. The signal generator can be controlled using SCPI commands that are output to a socket connection established in your program.

Before you can use sockets LAN, you must select the signal generator's sockets port number to use:

- Standard mode. Available on port 5025. Use this port for simple programming.
- TELNET mode. The telnet SCPI service is available on port 5023.

NOTE For backward compatibility, on the E8663B, ESG, and PSG, the signal generator also accepts references to the Telnet SCPI service at port 7777 and sockets SCPI service at port 7778.

Ports 7777 and 7778 are disabled on the Agilent MXG.

An example using sockets LAN is given in "LAN Programming Interface Examples" on page 112 of this programming guide.

Using Telnet LAN

Telnet provides a means of communicating with the signal generator over the LAN. The Telnet client, run on a LAN connected computer, will create a login session on the signal generator. A connection, established between computer and signal generator, generates a user interface display screen with `SCPI>` prompts on the command line.

Using the Telnet protocol to send commands to the signal generator is similar to communicating with the signal generator over GPIB. You establish a connection with the signal generator and then send or receive information using SCPI commands. Communication is interactive: one command at a time.

NOTE The Windows 2000 operating systems use a command prompt style interface for the Telnet client. Refer to the Figure 2-7 on page 44 for an example of this interface.

Windows XP operating systems and newer can use this section to better understand how to use the signal generator with port settings. For more information, refer to the help software of the IO libraries being used.

The following telnet LAN connections are discussed:

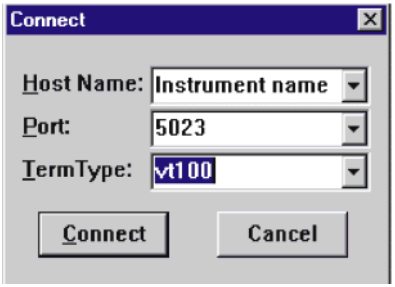
- "Using Telnet and MS–DOS Command Prompt" on page 41
- "Using Telnet On a PC With a Host/Port Setting Menu GUI" on page 42
- "Using Telnet On Windows 2000" on page 43
- "The Standard UNIX Telnet Command" on page 44

A Telnet example is provided in "Unix Telnet Example" on page 45.

Using Telnet and MS–DOS Command Prompt

1. On your PC, click **Start** > **Programs** > **Command Prompt**.
2. At the command prompt, type in `telnet`.
3. Press the **Enter** key. The Telnet display screen will be displayed.
4. Click on the **Connect** menu then select **Remote System**. A connection form (Figure 2-5) is displayed.

Figure 2-5 Connect Form (Agilent IO Library version J.01.0100)

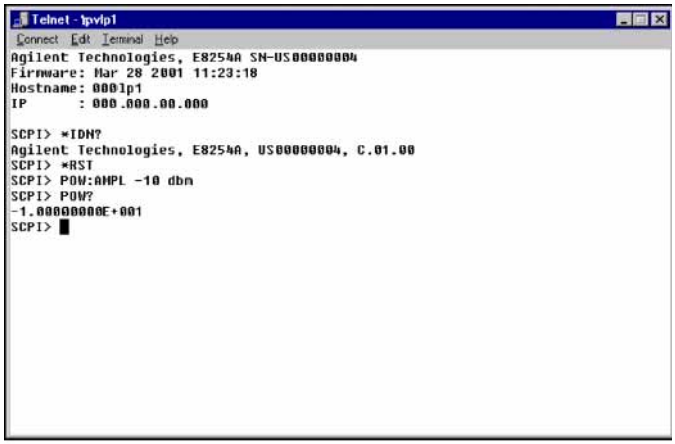


5. Enter the **hostname**, **port number**, and **TermType** then click **Connect**.
   - Host Name–IP address or hostname
   - Port–5023
   - Term Type–vt100
6. At the `SCPI>` prompt, enter SCPI commands. Refer to Figure 2-6 on page 43.
7. To signal device clear, press **Ctrl–C** on your keyboard.
8. Select **Exit** from the **Connect** menu and type `exit` at the command prompt to end the Telnet session.

Using Telnet On a PC With a Host/Port Setting Menu GUI

1. On your PC, click **Start** > **Run**.
2. Type `telnet` then click the **OK** button. The Telnet connection screen will be displayed.
3. Click on the **Connect** menu then select **Remote System**. A connection form is displayed. See Figure 2-5.
4. Enter the **hostname**, **port number**, and **TermType** then click **Connect**.
   - Host Name–signal generator's IP address or hostname
   - Port–5023
   - Term Type–vt100
5. At the `SCPI>` prompt, enter SCPI commands. Refer to Figure 2-6 on page 43.
6. To signal device clear, press **Ctrl–C**.
7. Select **Exit** from the **Connect** menu to end the Telnet session.

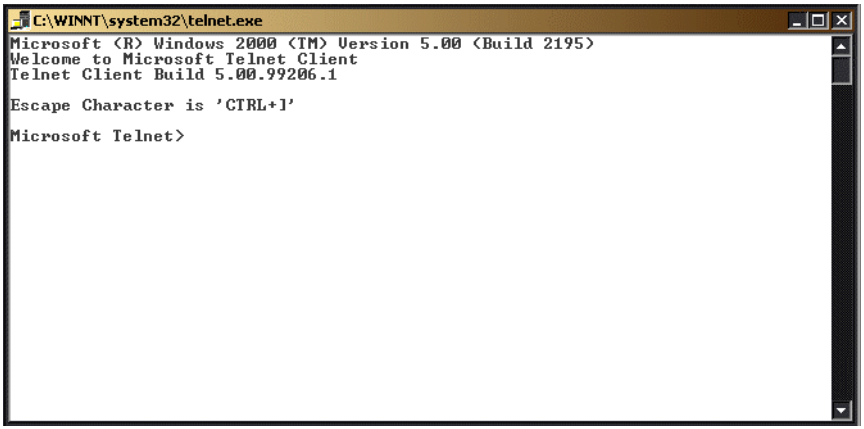
Figure 2-6 Telnet Window (Windows 2000)



Using Telnet On Windows 2000

1. On your PC, click **Start** > **Run**.
2. Type `telnet` in the run text box, then click the **OK** button. The Telnet connection screen will be displayed. See Figure 2-7 on page 44 (Windows 2000).
3. Type `open` at the prompt and then press the **Enter** key. The prompt will change to (`to`).
4. At the (`to`) prompt, enter the signal generator's IP address followed by a space and `5023`, which is the Telnet port associated with the signal generator.
5. At the `SCPI>` prompt, enter SCPI commands. Refer to commands shown in Figure 2-6 on page 43.
6. To escape from the SCPI> session type `Ctrl-]`.
7. Type `quit` at the prompt to end the Telnet session.

Figure 2-7 Telnet 2000 Window


C:\WINNT\system32\telnet.exe
Microsoft (R) Windows 2000 (TM) Version 5.00 (Build 2195)
Welcome to Microsoft Telnet Client
Telnet Client Build 5.00.99206.1
Escape Character is 'CTRL+]'
Microsoft Telnet>

The Standard UNIX Telnet Command

Synopsis

`telnet [host [port]]`

Description

This command is used to communicate with another host using the Telnet protocol. When the command `telnet` is invoked with `host` or `port` arguments, a connection is opened to the host, and input is sent from the user to the host.

Options and Parameters

The command `telnet` operates in character–at–a–time or line–by–line mode. In line–by–line mode, typed text is echoed to the screen. When the line is completed (by pressing the **Enter** key), the text line is sent to `host`. In character–at–a–time mode, text is echoed to the screen and sent to `host` as it is typed. At the UNIX prompt, type `man telnet` to view the options and parameters available with the `telnet` command.

NOTE If your Telnet connection is in line–by–line mode, there is no local echo. This means you cannot see the characters you are typing until you press the **Enter** key. To remedy this, change your Telnet connection to character–by–character mode. Escape out of Telnet, and at the `telnet>` prompt, type `mode char`. If this does not work, consult your Telnet program's documentation.

Unix Telnet Example

To connect to the instrument with host name `myInstrument` and port number `5023`, enter the following command on the command line: `telnet myInstrument 5023`.

When you connect to the signal generator, the UNIX window will display a welcome message and a SCPI command prompt. The instrument is now ready to accept your SCPI commands. As you type SCPI commands, query results appear on the next line. When you are done, break the Telnet connection using an escape character. For example, **`Ctrl-]`**,where the control key and the**]** are pressed at the same time. The following example shows Telnet commands:

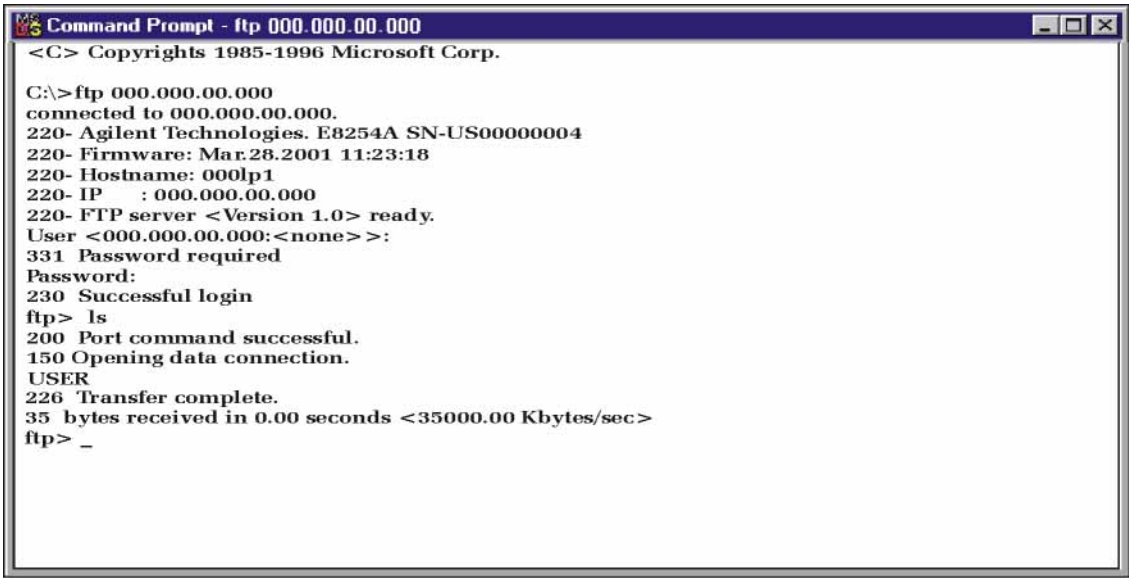
```
$ telnet myinstrument 5023
Trying....
Connected to signal generator
Escape character is '^]'.
Agilent Technologies, E44xx SN-US00000001
Firmware:
Hostname: your instrument
IP :xxx.xx.xxx.xxx
SCPI>
```

Using FTP

FTP allows users to transfer files between the signal generator and any computer connected to the LAN. For example, you can use FTP to download instrument screen images to a computer. When logged onto the signal generator with the FTP command, the signal generator's file structure can be accessed. Figure 2-8 shows the FTP interface and lists the directories in the signal generator's user level directory.

NOTE File access is limited to the signal generator's `/user` directory.

Figure 2-8 FTP Screen



ce917a

The following steps outline a sample FTP session from the MS–DOS Command Prompt:

1. On the PC click **Start** > **Programs** > **Command Prompt**.
2. At the command prompt enter:

   `ftp < IP address >` or `< hostname >`
3. At the user name prompt, press **enter**.
4. At the password prompt, press **enter**.

   You are now in the signal generator's user directory. Typing `help` at the command prompt will show you the FTP commands that are available on your system.
5. Type `quit` or `bye` to end your FTP session.
6. Type `exit` to end the command prompt session.

Using LXI

NOTE **LXI Class B Compliance Disclaimer**

As of this product firmware release in June 2008, LXI Class B Compliance Tests, using the new IEEE 158802008 Precision Time Protocol (PTP), were not available. This product provides the features of an LXI Class B instrument by adding LAN Triggering and Time Synchronization to its LXI Class C compliance. References to LXI Class B in the document and in the instrument menus and web pages do not insure LXI Class B compliance. This product will apply for LXI Class B Compliance when the LXI Consortium has approved its specification changes and can provide certified testing. Refer to *http://www.lxistandard.org/home*.

LXI–B is only available on instruments with serial prefixes ≥US/MY/SG4818.

This document assumes general familiarity with the LXI–C class of instruments and aims to clarify a number of general use cases for measurement synchronization between an MXA and an MXG using tools provided by the LXI–B standard. Refer to *http://www.lxistandard.org/home* and to *www.agilent.com/find/lxi*.

This LXI section contains the following:

- "IEEE 1588" on page 47
- "Peer to Peer Messaging" on page 49
- "Configuring LXI Triggers" on page 50
- "Using the LXI Event Log" on page 52
- "Setting up and executing a list sweep measurement" on page 52
- "Synchronize a List Sweep Measurement Between an MXA and an MXG using Peer to Peer Messages" on page 53
- "For More Information" on page 58

IEEE 1588

The IEEE 1588 standard defines a Precision Time Protocol (PTP) for synchronizing various clocks connected by Ethernet. This will give your measurement instrumentation a common sense of time. To ensure that the MXA and MXG are operating with synchronized PTP clocks, follow the procedure outlined in the table below.

Time Synchronization using an MXA Signal Analyzer and a MXG Signal Generator

To ensure that the MXA and MXG are operating with synchronized PTP clocks, follow this procedure:

1. Connect the MXA and MXG to the same Ethernet switch and the ensure that both instruments are configured for LAN operation.
2. Power on the MXA and MXG.

3. Send the following SCPI command to both the MXA and MXG:

   `:LXI:CLOCk:PTP:DOMain 1`

   This parameter may be any integer between 0 to 127. The default value is 0.

Verifying Time Synchronization

To verify that both instruments are running PTP, open the `Interactive LXI` tool from a PC which is connected to the same switch as the MXA and MXG. This program is bundled with the Agilent IO Libraries Suite.

1. Open Interactive LXI.

   From a PC connected to the same subnet as the instrument go to: **Start** > **All Programs** > **Agilent IO Libraries Suite** > **Utilities** > **Interactive LXI**.

2. Open the Timing menu.

   Click on the tab labeled **Timing**.

3. Choose an active domain.

   From the **Active Domain** pull down menu select the PTP domain entered in step 3.

4. Update the clocks.

   Click **Update Clock List**.

5. Check that the MXA and MXG are located in the expected domain.

   Click on the **Clock Names** listed in the Clock List, then examine the **Clock Information** panel for the IP Addresses of the MXA and MXG.

6. View the PTP LAN traffic.

   Click the **PTP Messages** tab and select the **Timing Messages** checkbox.

Peer to Peer Messaging

The MXA and MXG are capable of sending and receiving LXI specific LAN packets. The packets are configurable, and may be sent when various instrument events occur during a measurement or state recall. Each instrument event has an associated sense of 0 or 1 to indicate whether or not the event is active. The instrument events that can cause an MXG to send an LXI LAN packet are summarized in the table below.

Table 2-1 LXI LAN Packets

Instrument Event	Description
OperationComplete	Indicates whether or not an operation is underway. • For example, a single sweep will result in: • **OperationComplete** = 1 before the sweep. • **OperationComplete** = 0 during the sweep. • **OperationComplete** = 1 after the sweep.
Settling	Indicates the instrument is settling.
Sweeping	Indicates whether or not the instrument is currently performing a sweep.
Waiting For Trigger	Indicates whether or not the instrument is waiting for a trigger event before a measurement begins.

Enabling the LXI Event Subsystem

Due to the high priority response time demands of handling events, enabling the LXI event subsystem can adversely impact certain specifications such as frequency and power switching speed. This becomes more noticeable at higher rates of LXI event traffic. By default, the subsystem is disabled.

To enable the LXI event subsystem from the front panel:

- Press **Utility** > **More** > **LXI–B** > **LXI–B Enabled**

To enable or disable the LXI event subsystem from SCPI:

- `:LXI:STATe ON|OFF`

NOTE The LXI Event subsystem state is a persistent instrument state; it is *not* affected by normal preset operations.

Using the Front Panel to Configure LXI Events

The MXG can be configured to send output LAN events for the pre–defined LXI events via the front panel softkeys.

1. Press **Utility** > **More** > **LXI–B** > **Configure LXI Events** > **Configure LXI Output Events**
2. From the Output Events menu:

   Press **Select Source**. Choose the instrument status event to be used as a source for the output LAN event.
3. Press **Toggle Event State** to enable the highlighted output LAN event.

Using SCPI to configure an LXI Output Events

More configuration options and the ability to configure custom events are available to the system integrator through SCPI commands. The following procedure provides a minimum set of commands to configure an output LAN event. Refer to the *SCPI Command Reference*.

1. Reset the MXG:

   Send the following SCPI command:

   ```
   *RST
   ```
2. Set the LXI Output LAN Event's source:

   Send the following SCPI command:

   ```
   :LXI:EVENt:LAN:SOURce "LAN0", "Sweeping"
   ```
3. Enable the LXI Output Event:

   Send the following SCPI command:

   ```
   :LXI:EVENt:LAN:ENABled "LAN0", 1
   ```

Verifying LXI Output LAN Events using LXI Interactive

Peer to peer message transmission can be verified by running LXI Interactive from a PC on the same subnet as the MXA.

1. From a PC connected to the same subnet as the instrument, go to **Start** > **All Programs** > **Agilent IO Libraries Suite** > **Utilities** > **Interactive LXI**.
2. Click the **Start** button in the LXI Event Receive box.
3. Incoming LAN Events will be displayed in the text box.

   If the initial setup had been configured using the front panel as on page 49 or SCPI commands (page 50), two LAN Events are expected per sweep: the rising and falling edges of the LAN Event `"LAN0"`.

Configuring LXI Triggers

The MXA and MXG are capable of reacting to incoming LXI LAN Events. Both instruments may be triggered by the receipt of a peer to peer message.

Using the front panel to configure an LXI Trigger on the MXG

The MXG is capable of reacting to an incoming LXI LAN Event by treating it as a trigger. The following procedure describes how to set up the MXG sweep trigger to use an LXI event via the front panel soft keys.

1. Select the LXI LAN trigger as the source for sweep triggers.

   Press **Sweep** > **More** > **Sweep Trigger** > **More** > **LXI LAN**

2. Select a Trigger LAN Event.

   Press **Utility** > **More** > **LXI–B** > **Configure LXI Events** > **Configure LXI Trigger Events**

3. Enable the Trigger LAN Event.

   Press the **Toggle Event State** softkey.

Using SCPI to configure an LXI Trigger on the MXG

The MXG is capable of reacting to an incoming LXI LAN Event by treating it as a trigger. The following procedure describes how to set up the MXG sweep trigger to use an LXI event via SCPI.

1. Reset the MXG

   Send the following SCPI command:

   ```
   *RST
   ```

2. Select the LXI LAN trigger as the source for sweep triggers.

   Send the following SCPI command:

   ```
   :TRIGger:SOURce LAN3
   ```

3. Enable a Trigger LAN Event.

   Send the following SCPI command:

   ```
   :TRIGger:LXI:LAN:ENABled "LAN0",1
   ```

Verifying an LXI Trigger using LXI Interactive

Peer to peer message transmission can be verified by running LXI Interactive from a PC on the same subnet as the MXG.

1. Open Interactive LXI.

   From a PC connected to the same subnet as the instrument, go to **Start** > **All Programs** > **Agilent IO Libraries Suite** > **Utilities** > **Interactive LXI**.

2. Choose LAN Event.

   Select the desired LAN trigger from the **Event ID** dropdown menu.

3. Send LAN Event.

   Press the **Send** button. This puts the trigger out on the line.

4. Monitor Response as on page 50.

   This may be done assuming that the Output LAN event had been configured from the front panel (page 49) or SCPI commands (page 50).

Using the LXI Event Log

The LXI subsystem also provides an Event Log. The event log records all of the enabled LXI Event and Instrument Event activity and associates each action with an IEEE 1588 timestamp. Instrument Events are enabled by default, and will therefore appear in the log. Since many instrument events occur during every sweep, logging all of them may flood the event log. They may be disabled by sending the SCPI command:

```
:LXI:EVENt:STATus:ENABled "instrumentEvent",0
```

where the parameter `"instrumentEvent"` may be any event listed in Table 2-1. Note that the event must be re–enabled before being used as the source of an LXI Output LAN Event.

To view the LXI Event Log from the front panel, press the hard key System and then the softkeys Show, LXI, LXI Event Log. Once the Event Log is displayed, the arrow keys may be used to scroll through the list. The Event Log may also be queried via SCPI.

Setting up and executing a list sweep measurement

The following examples will illustrate procedures for setting up and executing a list sweep measurement using an MXA and an MXG synchronized with LXI.

Before launching into the procedure, it is important to understand the theory behind the method. The MXA and MXG use their LXI Output LAN Events to communicate internal state changes to each other. In the following synchronization, the rising edge of the `Waiting For Trigger` instrument event will be used to communicate to the other device that the next step may be executed.

For clarity, the MXG's `Waiting For Trigger` instrument event is mapped to the `LAN0` output event. Likewise, the MXA's `Waiting For Trigger` instrument event is mapped to the `LAN1` output event. The MXA is configured to trigger when it receives the `LAN0` event with a rising edge. The MXG is configured to trigger when it receives the `LAN1` event with a rising edge. Also, the `OperationComplete` output event is enabled, which has the `OperationComplete` instrument event mapped to it by default.

The MXG's list sweep is started first, causing it to source the first signal in its list. We then start the MXA's list sweep. The `OperationComplete` instrument event transitions low, causing an `OperationComplete` output event to go out on the LAN with a falling edge. The MXA takes a measurement at the first frequency in its list. When the MXA has finished its measurement, it moves on to the next entry in its list. The `Waiting For Trigger` instrument event transitions high, causing a `LAN1` output event to go out onto the LAN with a rising edge, and the MXA waits for its next trigger.

The MXG receives the `LAN1` event and triggers. The MXG's `Waiting For Trigger` instrument event transitions low, and a `LAN0` output event goes onto the LAN with a falling edge. The MXA takes no action, since it is configured to trigger only on rising edges. The MXG is now sourcing the next signal in its list. Once that signal has settled, it prepares to move on to its list's next entry. The MXG's `Waiting For Trigger` instrument event transitions high, and a `LAN0` output event goes onto the LAN with a rising edge. The MXG waits for its next trigger.

The MXA receives the `LAN0` event and triggers. The MXA's `Waiting For Trigger` instrument event transitions low, and a `LAN1` output event goes onto the LAN with a falling edge. The MXG takes no action, since it is configured to trigger only on rising edges. The MXA completes its measurement and prepares to move on to the next frequency in its list. The MXA's `Waiting For Trigger` instrument event transitions high, and a `LAN1` output event goes onto the LAN with a rising edge. The MXA waits for its next trigger.

The two instruments continue to step through their lists until the MXA has completed measurements at every frequency in its list. Once the final measurement has completed, the `OperationComplete` instrument event transitions high, causing the `OperationComplete` output event to go out onto the LAN with a rising edge. This event may be caught by a controller to signal the end of the measurement. The controller may then retrieve the list sweep measurement results from the MXA.

Synchronize a List Sweep Measurement Between an MXA and an MXG using Peer to Peer Messages

1. Disable LXI Output LAN Events on the MXG:

   Send the following SCPI command:

   `:LXI:EVENt:OUTPut:LAN:DISable:ALL`

2. Reset the MXG:

   Send the following SCPI command:

   `*RST`

3. Choose the MXG's PTP domain:

   Send the following SCPI command:

   `:LXI:CLOCk:PTP:DOMain 0`

   The parameter may be any integer between 0 to 127.

4. Choose the MXG's LXI domain:

   Send the following SCPI command:

   `:LXI:EVENt:DOMain 0`

   The parameter may be any integer between 0 to 127.

5. Preset the MXG's list sweep:

   Send the following SCPI command:

   `:LIST:TYPE:LIST:INIT:PRES`

6. Sets the MXG to expect a list of frequencies:

   Send the following SCPI command:

   `:FREQ:MODE LIST`

7. Sets the MXG to expect a list of powers:

   Send the following SCPI command:

   `:POW:MODE LIST`

8. Sets the MXG how the lists will be entered:

   Send the following SCPI command:

   `:LIST:TYPE LIST`

   An arbitrary list will be used instead of range and step size arguments.

9. Send the MXG a list of frequencies:

   Send the following SCPI command:

   ```
   :LIST:FREQ 100MHz,200MHz,300MHz,400MHz,500MHz
   ```

   The MXG will put out signals at these frequencies and in this order.

10. Send the MXG a list of powers:

    Send the following SCPI command:

    ```
    :LIST:POW 0dBm,-1dBm,-2dBm,-3dBm,-4dBm
    ```

    The MXG will put out signals at these powers and in this order.

11. Turn off MXG signal modulation:

    Send the following SCPI command:

    ```
    :OUTP:MOD OFF
    ```

12. Turn MXG signal output on:

    Send the following SCPI command:

    ```
    :OUTP ON
    ```

13. Configure the MXG's LXI Output LAN Event:

    Send the following SCPI commands:

    ```
    :LXI:EVENt:OUTPut:LAN:DRIVe "LAN0",NORMal
    ```

    ```
    :LXI:EVENt:OUTPut:LAN:SLOPe "LAN0",POS
    ```

    ```
    :LXI:EVENt:OUTPut:LAN:SOURce "LAN0","Waiting For Trigger"
    ```

    Together, these commands will cause the MXG to send a `"LAN0"` peer to peer message every time the `"Waiting For Trigger"` instrument event changes state.

14. Set the MXG's trigger source:

    Send the following SCPI command:

    ```
    :LIST:TRIG:SOUR LAN
    ```

15. Enable the MXG's LXI LAN Event trigger:

    Send the following SCPI command:

    ```
    :TRIG:LXI:LAN:ENABled "LAN1",1
    ```

16. Enable the MXG's LXI Output LAN Event:

    Send the following SCPI command:

    ```
    :LXI:EVENt:OUTPut:LAN:ENABled "LAN0",1
    ```

17. Disable LXI Output LAN Events on the MXA:

    Send the following SCPI command:

    ```
    :LXI:EVENt:OUTPut:LAN:DISable:ALL
    ```

18. Put the MXA into SA mode:

    Send the following SCPI command:

    ```
    :INST:SEL SA
    ```

19. Put the MXA into single sweep mode:

    Send the following SCPI command:

    ```
    :INIT:CONT OFF
    ```

20. Choose the MXA's PTP domain:

    Send the following SCPI command:

    ```
    :LXI:CLOCk:PTP:DOMain 0
    ```

    The parameter value should match the one used in step 3.

21. Choose the MXA's LXI domain:

    Send the following SCPI command:

    ```
    :LXI:EVENt:DOMain 0
    ```

    The parameter value should match the one used in step 4.

22. Move the MXA into the list sweep measurement:

    Send the following SCPI command:

    ```
    :CONF:LIST
    ```

23. Send the MXA a list of frequencies:

    Send the following SCPI command:

    ```
    :LIST:FREQ 100MHz,200MHz,300MHz,400MHz,500MHz
    ```

    The MXA will put out signals at these frequencies and in this order.

24. Configure the MXA's list sweep parameters:

    Send the following SCPI commands:

    ```
    :LIST:ATT 10dB
    :LIST:BAND:RES:TYPE FLAT
    :LIST:BAND:RES 300kHz
    :LIST:BAND:VID 3MHz
    :LIST:SWE:TIME 1e-6 s
    :LIST:TRIG:DEL 0
    :LIST:DET RMS
    ```

25. Make sure the MXA's LXI LAN triggers are disabled:

    Send the following SCPI command:

    ```
    :TRIG:LXI:LAN:DISable:ALL
    ```

26. Set the MXA's trigger source to LXI LAN:

    Send the following SCPI command:

    ```
    :LIST:TRIG:SOUR LAN
    ```

27. Configure the MXA's LXI Output LAN Event:

    Send the following SCPI commands:

    ```
    :LXI:EVENt:OUTPut:LAN:DRIVe "LAN1",NORMal
    ```

    ```
    :LXI:EVENt:OUTPut:LAN:SLOPe "LAN1",POS
    ```

    ```
    :LXI:EVENt:OUTPut:LAN:SOURce "LAN1","WaitingForTrigger"
    ```

    This will cause the MXA to take a measurement at the first frequency in its list. It will also begin waiting for the `"LAN0"` peer to peer message before moving on to the next measurement.

28. Enable the MXA's LXI LAN Event trigger:

    Send the following SCPI commands:

    ```
    :TRIG:LXI:LAN:ENABled "LAN0",1
    ```

29. Enable the MXA's `"OperationComplete"` LXI Output LAN Event:

    Send the following SCPI commands:

    ```
    :LXI:EVENt:OUTPut:LAN:ENABled "OperationComplete",1
    ```

    The MXA will send an `"OperationComplete"` peer to peer message when the `"OperationComplete"` instrument event changes state.

30. Begin the MXG's list sweep:

    Send the following SCPI commands:

    ```
    :INIT:CONT ON
    ```

    This will cause the MXG to put out a signal at the first frequency in its list. It will also begin waiting for the `"LAN1"` peer to peer message before moving on to the next signal.

31. Begin the MXA's list sweep:

    Send the following SCPI commands:

    ```
    :INIT:LIST
    ```

    This will cause the MXA to take a measurement at the first frequency in its list. It will also begin waiting for the `"LAN0"` peer to peer message before moving on to the next measurement.

32. Enable the MXA's `"LAN1"` LXI Output LAN Event:

    Send the following SCPI commands:

    ```
    :LXI:EVENt:OUTPut:LAN:ENABled "LAN1",1
    ```

    The `"LAN1"` peer to peer message is now enabled.

33. Send the MXG a `"LAN1"` peer to peer message:

    Using *Interactive LXI* or the *Agilent IO Libraries TMFramework LXI library*: send a `"LAN1"` peer to peer message to the MXG.

    This will start the synchronization sequence.

34. The MXA waits for the `"OperationComplete"` instrument event:

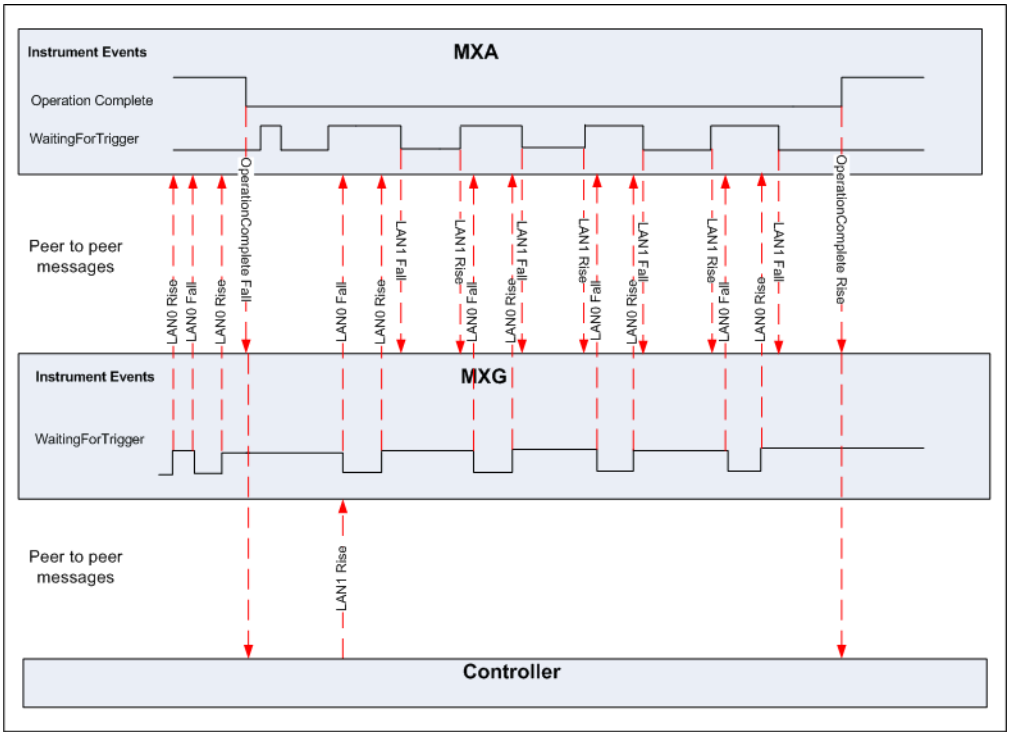
    Detect the peer to peer traffic using *Interactive LXI*. To programmatically listen for LXI peer to peer messages, use the *Agilent IO Libraries TMFramework LXI library*.

    When the MXA has completed its list, it will send the peer to peer message `"OperationComplete"` with a rising edge to MXG.

35. Send query to MXA for the measurement results:

    ```
    :FETCh:LIST?
    ```

Figure 2-9 Illustration of a List Sweep Measurement Between an MXA and an MXG using Peer to Peer Messages.



For More Information

For more information on using LXI see the Agilent website dedicated to LXI instrumentation: *www.agilent.com/find/lxi*.

Agilent LXI Application Notes:

- Using LXI to go beyond GPIB, PXI and VXI (AN 1465–20)
- 10 Good Reasons to Switch to LXI (AN 1465–21)
- Transitioning from GPIB to LXI (AN 1465–22)
- How to Use VXI and PXI in Your New LXI Test System (AN 1465–23)
- Using Synthetic Instruments in Your Test System (AN 1465–24)
- Migrating system software from GPIB to LAN/LXI (AN 1465–25)
- Modifying a GPIB System to Include LAN/LXI (AN 1465–26)

Using RS–232 (ESG, PSG, and E8663B Only)

CAUTION For long strings of commands and waveform downloads, upgrading to Agilent IO Libraries 15.0 and above *decreases* RS–232 performance, resulting in an Error –310.

NOTE Because GPIB, LAN, and USB offer better communication performance, Agilent recommends that RS–232 only be used for interactive sessions or short commands.

The RS–232 serial interface is available on the ESG signal generators.

The PSG and E8663B's **AUXILIARY INTERFACE** connector is compatible with ANSI/EIA232 (RS–232) serial connection but GPIB and LAN are recommended for making faster measurements and when downloading files. Refer to the *User's Guide*.

The RS–232 serial interface can be used to communicate with the signal generator. The RS–232 connection was once standard on most PCs but has now been replaced by USB. RS–232 can be connected to the signal generator's rear panel connector using the cable described in Table 2-2 on page 61. Many functions provided by GPIB, with the exception of indefinite blocks, parallel polling, serial polling, GET, non–SCPI remote languages, SRQ, and remote mode are available using the RS–232 interface.

The serial port sends and receives data one bit at a time, therefore RS–232 communication is slow. The data transmitted and received is usually in ASCII format with SCPI commands being sent to the signal generator and ASCII data returned.

The following sections contain information on selecting and connecting IO libraries and RS–232 interface hardware on the signal generator to a computer's RS–232 connector.

- "Selecting IO Libraries for RS–232" on page 59
- "Setting Up the RS–232 Interface" on page 60
- "Verifying RS–232 Functionality" on page 62

Selecting IO Libraries for RS–232

The IO libraries can be downloaded from the National Instrument website, *http://www.ni.com*, or Agilent's website, *http://www.agilent.com*. The following is a discussion on these libraries.

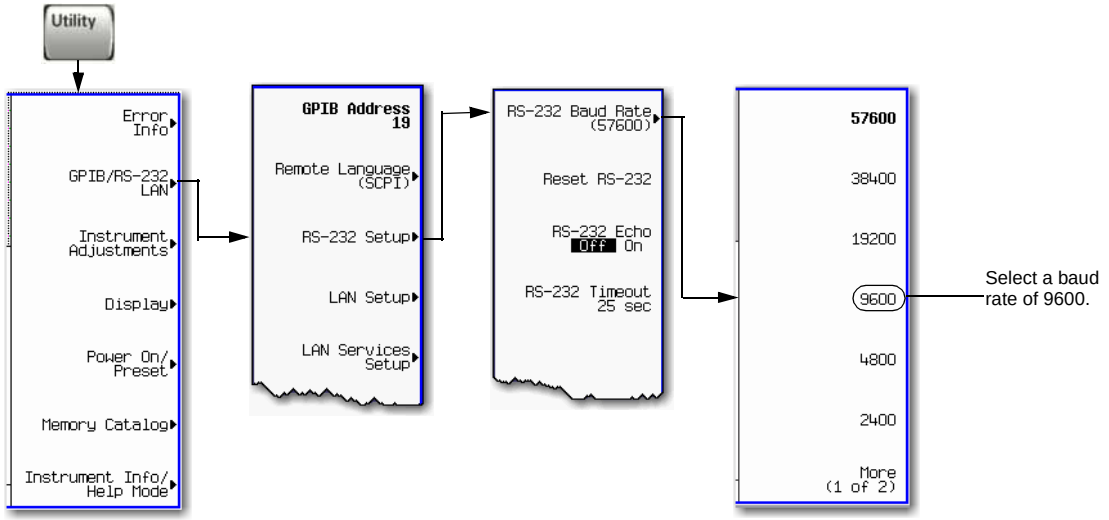
CAUTION For long strings of commands and waveform downloads, upgrading to Agilent IO Libraries 15.0 and above *decreases* RS–232 performance, resulting in an Error –310.

Because of the potential for portability problems, running Agilent SICL without the VISA overlay is *not* recommended by Agilent Technologies.

HP Basic	The HP Basic language has an extensive IO library that can be used to control the signal generator over the RS–232 interface. This library has many low level functions that can be used in BASIC applications to control the signal generator over the RS–232 interface.
VISA	VISA is an IO library used to develop IO applications and instrument drivers that comply with industry standards. It is recommended that the VISA library be used for programming the signal generator. The NI–VISA and Agilent VISA libraries are similar implementations of VISA and have the same commands, syntax, and functions. The differences are in the lower level IO libraries used to communicate over the RS–232; NI–488.2 and SICL respectively.
NI–488.2	NI–488.2 IO libraries can be used to develop applications for the RS–232 interface. See National Instrument's website for information on NI–488.2.
SICL	Agilent SICL can be used to develop applications for the RS–232 interface. See Agilent's website for information on SICL.

Setting Up the RS–232 Interface

1. Setting the RS–232 Interface Baud Rate (ESG/PSG/E8663B)



SCPI commands:

```
:SYSTem:COMMunicate:SERial:BAUD <number>
:SYSTem:COMMunicate:SERial:BAUD?
```

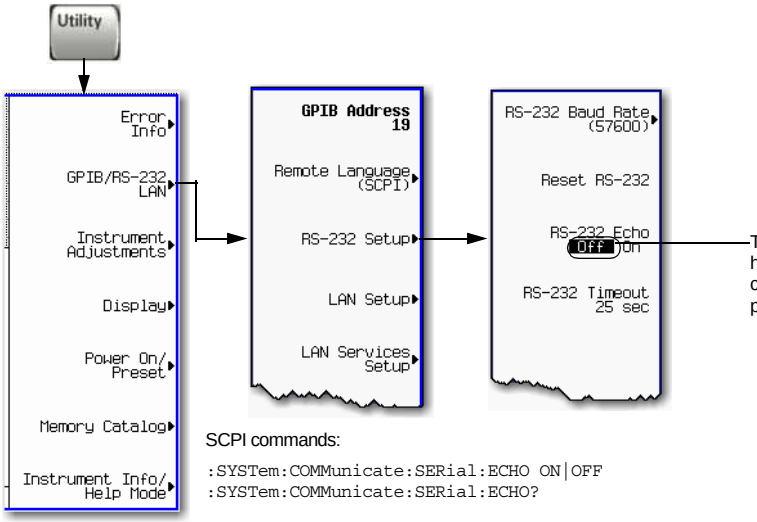
For details on each key, use the key help (described in *User's Guide*). For additional SCPI command information, refer to the SCPI Command Reference.

NOTE Configure your computer to use baud rates 57600 or lower only. Select the signal generator's baud rate to match the baud rate of your computer or UNIX workstation or adjust the baud rate settings on your computer to match the baud rate setting of the signal generator.

The default baud rate for VISA is 9600. This baud rate can be changed with the "VI_ATTR_ASRL_BAUD" VISA attribute.

---

2. Setting the RS–232 Echo Softkey



For details on each key, use the key help (described in *User's Guide*). For additional SCPI command information, refer to the SCPI Command Reference.

3. Connect an RS–232 cable from the computer's serial connector to the ESG signal generator's **RS–232** connector or the PSG or E8663B's **AUXILIARY INTERFACE** connector. Refer to Table 2-2 for RS–232 cable information.

Table 2-2 RS–232 Serial Interface Cable

Quantity	Description	Agilent Part Number
1	Serial RS–232 cable 9–pin (male) to 9–pin (female)	8120–6188

NOTE Any 9 pin (male) to 9 pin (female) straight–through cable that directly wires pins 2, 3, 5, 7, and 8 may be used.

Verifying RS–232 Functionality

You can use the HyperTerminal program available on your computer to verify the RS–232 interface functionality. To run the HyperTerminal program, connect the RS–232 cable between the computer and the signal generator and perform the following steps:

1. On the PC click **Start** > **Programs** > **Accessories** > **Communications** > **HyperTerminal**.
2. Select **HyperTerminal**.
3. Enter a name for the session in the text box and select an icon.
4. Select **COM1** (COM2 can be used if COM1 is unavailable).
5. In the COM1 (or COM2, if selected) properties, set the following parameters:
   - Bits per second: 9600 must match signal generator's baud rate; for more information, refer to "Setting Up the RS–232 Interface" on page 60.
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
   - Flow Control: None

NOTE Flow control, via the RTS line, is driven by the signal generator. For the purposes of this verification, the controller (PC) can ignore this if flow control is set to None. However, to control the signal generator programmatically or download files to the signal generator, you *must* enable RTS–CTS (hardware) flow control on the controller. Note that only the RTS line is currently used.

Software Flow Control using XON and XOFF is not supported. Only RTS–CTS hardware flow is supported.

---

6. Go to the HyperTerminal window and select **File** > **Properties**.
7. Go to **Settings** > **Emulation** and select **VT100**.
8. Leave the **Backscroll buffer lines** set to the default value.
9. Go to **Settings** > **ASCII Setup**.
10. Check the first two boxes and leave the other boxes as default values.

Once the connection is established, enter the SCPI command `*IDN?` followed by `<Ctrl j>` in the HyperTerminal window. The `<Ctrl j>` is the new line character (on the keyboard press the **Cntrl** key and the **j** key simultaneously).

The signal generator should return a string similar to the following, depending on model:

Agilent Technologies <*instrument model name and number*>, US40000001,C.02.00

Character Format Parameters

The signal generator uses the following character format parameters when communicating via RS–232:

- Character Length: Eight data bits are used for each character, excluding start, stop, and parity bits.
- Parity Enable: Parity is disabled (absent) for each character.
- Stop Bits: One stop bit is included with each character.

If You Have Problems

1. Verify that the baud rate, parity, and stop bits are the same for the computer and signal generator.
2. Verify that the RS–232 cable is identical to the cable specified in Table 2-2.
3. Verify that the application is using the correct computer COM port and that the RS–232 cable is properly connected to that port.
4. Verify that the controller's flow control is set to RTS–CTS.

RS–232 Programming Interface Examples

NOTE The portions of the programming examples discussed in this section are taken from the full text of these programs that can be found in Chapter 3, "Programming Examples."

- "Interface Check Using HP BASIC" on page 64
- "Interface Check Using VISA and C" on page 64
- "Queries Using HP Basic and RS–232" on page 65
- "Queries for RS–232 Using VISA and C" on page 65

Before Using the Examples

Before using the examples: On the signal generator select the following settings:

- Baud Rate – 9600 must match computer's baud rate
- RS–232 Echo – Off

The following sections contain HP Basic and C lines of programming removed from the programming interface examples in Chapter 3, Programming Examples,, these portions of programming demonstrate the important features to consider when developing programming for use with the RS–232 interface.

NOTE For RS–232 programming examples, refer to "RS-232 Programming Interface Examples (ESG/PSG/E8663B Only)" on page 142.

Interface Check Using HP BASIC

This portion of the example program "Interface Check Using HP BASIC" on page 64, causes the signal generator to perform an instrument reset. The SCPI command `*RST` will place the signal generator into a pre–defined state.

The serial interface address for the signal generator in this example is 9. The serial port used is COM1 (Serial A on some computers). Refer to "Using RS–232 (ESG, PSG, and E8663B Only)" on page 58 for more information.

The following program example is available on the signal generator's Documentation CD–ROM as rs232ex1.txt. For the full text of this program, refer to "Interface Check Using HP BASIC" on page 142 or to the signal generator's documentation CD–ROM.

```
170     CONTROL 9,0;1        ! Resets the RS-232 interface
180     CONTROL 9,3;9600     ! Sets the baud rate to match the sig gen
190     STATUS 9,4;Stat      ! Reads the value of register 4
200     Num=BINAND(Stat,7)   ! Gets the AND value
210     CONTROL 9,4;Num      ! Sets parity to NONE
220     OUTPUT 9;"*RST"      ! Outputs reset to the sig gen
```

Interface Check Using VISA and C

This portion of the example program "Interface Check Using VISA and C" on page 64, uses VISA library functions to communicate with the signal generator. The program verifies that the RS–232 connections and interface are functional. In this example the COM2 port is used. The serial port is referred to in the VISA library as 'ASRL1' or 'ASRL2' depending on the computer serial port you are using.

The following program example is available on the signal generator Documentation CD–ROM as rs232ex1.cpp. For the full text of this program, refer to "Interface Check Using VISA and C" on page 143 or to the signal generator's documentation CD–ROM.

```
int baud=9600;// Set baud rate to 9600

ViSession defaultRM, vi;// Declares a variable of type ViSession
// for instrument communication on COM 2 port
ViStatus viStatus = 0;
                        // Opens session to RS-232 device at serial port 2
viStatus=viOpenDefaultRM(&defaultRM);
viStatus=viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &vi);


viStatus=viEnableEvent(vi, VI_EVENT_IO_COMPLETION, VI_QUEUE,VI_NULL);


viClear(vi);// Sends device clear command
// Set attributes for the session
viSetAttribute(vi,VI_ATTR_ASRL_BAUD,baud);
viSetAttribute(vi,VI_ATTR_ASRL_DATA_BITS,8);
```

Queries Using HP Basic and RS–232

This portion of the example program "Queries Using HP Basic and RS–232" on page 65, example program demonstrates signal generator query commands over RS–232. Query commands are of the type `*IDN?` and are identified by the question mark that follows the mnemonic.

Start HP Basic, type in the following commands, and then RUN the program:

The following program example is available on the signal generator Documentation CD–ROM as `rs232ex2.txt`. For the full text of this program, refer to "Queries Using HP Basic and RS-232" on page 145 or to the signal generator's documentation CD–ROM.

```
190    OUTPUT 9;"*IDN?"           ! Querys the sig gen ID
200    ENTER 9;Str$               ! Reads the ID
210    WAIT 2                     ! Waits 2 seconds
220    PRINT "ID =",Str$          ! Prints ID to the screen
230    OUTPUT 9;"POW:AMPL -5 dbm" ! Sets the the power level to -5 dbm
240    OUTPUT 9;"POW?"            ! Querys the power level of the sig gen
```

Queries for RS–232 Using VISA and C

This portion of the example program "Queries for RS–232 Using VISA and C" on page 65, uses VISA library functions to communicate with the signal generator. The program verifies that the RS–232 connections and interface are functional.

The following program example is available on the signal generator Documentation CD–ROM as `rs232ex2.cpp`. For the full text of this program, refer to "Queries for RS-232 Using VISA and C" on page 146 or to the signal generator's documentation CD–ROM.

```
status = viOpenDefaultRM(&defaultRM);// Initializes the system
// Open communication with Serial Port 2
status = viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &instr);
```

Using USB (Agilent MXG)

CAUTION USB cables are not industrial graded and potentially allows data loss in noisy environments.

USB cables do not have a latching mechanism and the cables can be pulled out of the PC or instrument relatively easily.

The maximum length for USB cables is 30 m, including the use of inline repeaters.

NOTE The USB interface is available only on the Agilent MXG signal generator.

The Agilent MXG's USB 2.0 interface supports USBTMC or USBTMC–USB488 specifications.

For more information on connecting instruments to the USB, refer to the Agilent Connection Expert in the Agilent IO Libraries Help.

USB 2.0 connectors can be used to communicate with the signal generator. The N5161A/62A/81A/82A is equipped with a Mini–B 5 pin rear panel connector (device USB). Use a Type–A to Mini–USB 5 pin cable to connect the signal generator to the computer (Refer to "Setting Up the USB Interface" on page 67). Connect the Type–A front panel connector (host USB) can be used to connect a mouse, a keyboard, or a USB 1.1/2.0 flash drive (USB media). (Refer to the User's Guide.) ARB waveform encryption of proprietary information is supported. Many functions provided by GPIB, including GET, non–SCPI remote languages, and remote mode are available using the USB interface.

NOTE For a list of compatible flash drives to use with the USB external interface. Refer to *http://www.agilent.com/find/mxg*.

Do not *use* the Type A front panel USB to connect to a computer.

The following sections contain information on selecting and connecting I/O libraries and the USB interface that are required to remotely program the signal generator via computer and combining those choices with one of several possible USB interface protocols.

- "Selecting I/O Libraries for USB" on page 67
- "Setting Up the USB Interface" on page 67
- "Verifying USB Functionality" on page 68

Selecting I/O Libraries for USB

CAUTION The Agilent MXG's USB interface requires Agilent IO Libraries Suite 14.1 or newer to run properly. For more information on connecting instruments to the USB, refer to the Agilent Connection Expert in the Agilent IO Libraries Help.

The I/O libraries can be downloaded from the National Instrument website, *http://www.ni.com*, or Agilent's website, *http://www.agilent.com*. The following is a discussion on these libraries.

NOTE I/O applications such as IVI–COM or VXI*plug&play* can be used in place of VISA.

VISA — VISA is an I/O library used to develop I/O applications and instrument drivers that comply with industry standards. It is recommended that the VISA library be used for programming the signal generator. The NI–VISA and Agilent VISA libraries are similar implementations of VISA and have the same commands, syntax, and functions. The differences are in the lower level I/O libraries used to communicate over the USB; NI–488.2 and SICL respectively.

NI–488.2 — NI–488.2 I/O libraries can be used to develop applications for the USB interface. See National Instrument's website for information on NI–488.2.

SICL — Agilent SICL can be used to develop applications for the USB interface. See Agilent's website for information on SICL.

CAUTION Because of the potential for portability problems, running Agilent SICL without the VISA overlay is *not* recommended by Agilent Technologies.

Setting Up the USB Interface

Rear Panel Interface (Mini–B 5 pin)

To use USB, connect the USB cable (Refer to Table 2-3, "USB Interface Cable," on page 67, for USB cable information.) between the computer and the signal generator's rear panel Mini–B 5–pin USB connector.

Table 2-3 USB Interface Cable

Quantity	Description	Agilent Part Number
1	USB cable Mini–B 5 pin to Type–A	82357–61601

Front Panel USB (Type–A)

For details on using the front panel USB (Type–A) and the front panel USB Media operation, refer to the *User's Guide*.

Verifying USB Functionality

Mini–B 5 Pin Rear Panel Connector

NOTE For information on verifying your Mini–B 5 pin USB (rear panel) functionality, refer to the Agilent Connection Expert in the Agilent IO Libraries Help. The Agilent IO libraries are included with your signal generator or Agilent GPIB interface board, or they can be downloaded from the Agilent website: *http://www.agilent.com*.

Type–A Front Panel USB Connector

For details on using the front panel USB (Type–A) and the front panel USB Media operation, refer to the *User's Guide*.

3 Programming Examples

NOTE For the N5161A/62A the softkey menus and features mentioned in this chapter are only available through the Web-Enabled MXG or via SCPI commands. Refer to "Using the Web Browser" on page 11 and to the *SCPI Command Reference*.

- Using the Programming Interface Examples on page 69
- GPIB Programming Interface Examples on page 74
- LAN Programming Interface Examples on page 112
- RS-232 Programming Interface Examples (ESG/PSG/E8663B Only) on page 142

Using the Programming Interface Examples

The programming examples for remote control of the signal generator use the GPIB, LAN, and RS-232 interfaces and demonstrate instrument control using different IO libraries and programming languages. Many of the example programs in this chapter are interactive; the user will be prompted to perform certain actions or verify signal generator operation or functionality. Example programs are written in the following languages:

HP Basic	C#
C/C++	Microsoft Visual Basic 6.0
Java	MATLAB
Perl	

These example programs are also available on the signal generator Documentation CD-ROM, enabling you to cut and paste the examples into a text editor.

NOTE The example programs set the signal generator into remote mode; front panel keys, except the Agilent MXG **Local/Esc/Cancel** or the ESG, PSG, and E8663B's **Local** key, are disabled. Press the Agilent MXG **Local/Esc/Cancel** or the ESG, PSG, and E8663B's **Local** key to revert to manual operation.

To have the signal generator's front panel update with changes caused by remote operations, enable the signal generator's Update in Remote function.

NOTE The Update in Remote function will slow test execution. For faster test execution, disable the Update in Remote function. (For more information, refer to or "Configuring the Display for Remote Command Setups (Agilent MXG)" on page 17.) or "Configuring the Display for Remote Command Setups (ESG/PSG/E8663B)" on page 18.

Programming Examples Development Environment

The C/C++ examples were written using an IBM-compatible personal computer (PC), configured as follows:

- Pentium® processor (Pentium is a registered trademark of Intel Corporation.)
- Windows NT 4.0 operating system or later
- C/C++ programming language with the Microsoft Visual C++ 6.0 IDE
- National Instruments PCI-GPIB interface card or Agilent GPIB interface card
- National Instruments VISA Library or Agilent VISA library
- COM1 or COM2 serial port available
- LAN interface card

The HP Basic examples were run on a UNIX 700 series workstation.

Running C++ Programs

When using Microsoft Visual C++ 6.0 to run the example programs, include the following files in your project.

When using the VISA library:

- add the visa32.lib file to the Resource Files
- add the visa.h file to the Header Files

When using the NI-488.2 library:

- add the GPIB-32.OBJ file to the Resource Files
- add the windows.h file to the Header Files
- add the Deci-32.h file to the Header Files

For information on the NI-488.2 library and file requirements refer to the National Instrument website. For information on the VISA library see the Agilent website or National Instrument's website.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the DHCP LAN (Agilent MXG)" on page 33 and "Configuring the VXI–11 for LAN (ESG/PSG/E8663B)" on page 31.

C/C++ Examples

- "Interface Check for GPIB Using VISA and C" on page 80
- "Queries for RS-232 Using VISA and C" on page 146
- "Local Lockout Using NI-488.2 and C++" on page 82
- "Queries Using NI-488.2 and Visual C++" on page 85
- "Queries for GPIB Using VISA and C" on page 87
- "Generating a CW Signal Using VISA and C" on page 89
- "Generating an Externally Applied AC-Coupled FM Signal Using VISA and C" on page 91
- "Generating an Internal FM Signal Using VISA and C" on page 93
- "Generating a Step-Swept Signal Using VISA and C++" on page 95
- "Reading the Data Questionable Status Register Using VISA and C" on page 101
- "Reading the Service Request Interrupt (SRQ) Using VISA and C" on page 105
- "VXI-11 Programming Using SICL and C++" on page 113
- "VXI-11 Programming Using VISA and C++" on page 114
- "Sockets LAN Programming and C" on page 116
- "Interface Check Using VISA and C" on page 143
- "Queries for RS-232 Using VISA and C" on page 146

Running C# Examples

To run the example program *State_Files.cs* on page 337, you must have the .NET framework installed on your computer. You must also have the Agilent IO Libraries installed on your computer. The .NET framework can be downloaded from the Microsoft website. For more information on running C# programs using .NET framework, see Chapter 6.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the VXI–11 for LAN (Agilent MXG)" on page 30 and "Configuring the VXI–11 for LAN (ESG/PSG/E8663B)" on page 31.

Running Basic Examples

The BASIC programming interface examples provided in this chapter use either HP Basic or Visual Basic 6.0 languages.

Visual Basic 6.0 Programming Examples

To run the example programs written in Visual Basic 6.0 you must include references to the IO Libraries. For more information on VISA and IO libraries, refer to the Agilent VISA User's Manual, available on Agilent's website: *http://www.agilent.com*. In the Visual Basic IDE (Integrated Development Environment) go to Project–References and place a check mark on the following references:

- Agilent VISA COM Resource Manager 1.0
- VISA COM 1.0 Type Library

NOTE If you want to use VISA functions such as viWrite, then you must add the visa32.bas module to your Visual Basic project.

The signal generator's VXI-11 SCPI service must be on before you can run the Download Visual Basic 6.0 programming example.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the VXI–11 for LAN (Agilent MXG)" on page 30 and "Configuring the VXI–11 for LAN (ESG/PSG/E8663B)" on page 31.

You can start a new Standard EXE project and add the required references. Once the required references are included, you can copy the example programs into your project and add a command button to `Form1` that will call the program.

The example Visual Basic 6.0 programs are available on the signal generator Documentation CD-ROM, enabling you to cut and paste the examples into your project.

Visual Basic Examples

The Visual Basic examples enable the use of waveform files and are located in Chapter 5.

- "Creating I/Q Data–Little Endian Order" on page 275
- "Downloading I/Q Data" on page 277

HP Basic Examples

- "Interface Check using HP Basic and GPIB" on page 78
- "Local Lockout Using HP Basic and GPIB" on page 81
- "Queries Using HP Basic and GPIB" on page 84
- "Queries Using HP Basic and RS-232" on page 145
- "Using 8757D Pass-Thru Commands (PSG with Option 007 Only)" on page 109

Running Java Examples

The Java program "Sockets LAN Programming Using Java" on page 139, connects to the signal generator via sockets LAN. This program requires Java version 1.1 or later be installed on your PC. For more information on sockets LAN programming with Java, refer to "Sockets LAN Programming Using Java" on page 139.

Running MATLAB Examples

For information regarding programming examples and files required to create and play waveform files, refer to Chapter 5.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the VXI–11 for LAN (Agilent MXG)" on page 30 and "Configuring the VXI–11 for LAN (ESG/PSG/E8663B)" on page 31.

Running Perl Examples

The Perl example "Sockets LAN Programming Using PERL" on page 141, uses PERL script to control the signal generator over the sockets LAN interface.

Using GPIB

GPIB enables instruments to be connected together and controlled by a computer. GPIB and its associated interface operations are defined in the ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1992. See the IEEE website, *http://www.ieee.org*, for details on these standards.

The following sections contain information for installing a GPIB interface card or NI-GPIB interface card for your PC or UNIX-based system.

- "Installing the GPIB Interface Card" on page 73

For more information on setting up a GPIB interface card or NI-GPIB interface card, refer to:

- "Set Up the GPIB Interface" on page 25
- "Verify GPIB Functionality" on page 26

NOTE You can also connect GPIB instruments to a PC USB port using the Agilent 82357A USB/GPIB Interface Converter, which eliminates the need for a GPIB card. For more information, go to *http://www.agilent.com/find/gpib*.

Installing the GPIB Interface Card

Refer to "Installing the GPIB Interface" on page 23.

GPIB Programming Interface Examples

- "Interface Check using HP Basic and GPIB" on page 78
- "Interface Check Using NI-488.2 and C++" on page 79
- "Interface Check for GPIB Using VISA and C" on page 80
- "Local Lockout Using HP Basic and GPIB" on page 81
- "Local Lockout Using NI-488.2 and C++" on page 82
- "Queries Using HP Basic and GPIB" on page 84
- "Queries Using NI-488.2 and Visual C++" on page 85
- "Queries for GPIB Using VISA and C" on page 87
- "Generating a CW Signal Using VISA and C" on page 89
- "Generating an Externally Applied AC-Coupled FM Signal Using VISA and C" on page 91
- "Generating an Internal FM Signal Using VISA and C" on page 93
- "Generating a Step-Swept Signal Using VISA and C++" on page 95
- "Generating a Swept Signal Using VISA and Visual C++" on page 96
- "Saving and Recalling States Using VISA and C" on page 98
- "Reading the Data Questionable Status Register Using VISA and C" on page 101
- "Reading the Service Request Interrupt (SRQ) Using VISA and C" on page 105
- "Using 8757D Pass-Thru Commands (PSG with Option 007 Only)" on page 109

Before Using the GPIB Examples

HP Basic addresses the signal generator at 719. The GPIB card is addressed at 7 and the signal generator at 19. The GPIB address designator for other libraries is typically GPIB0 or GPIB1.

GPIB Function Statements (Command Messages)

Function statements are the basis for GPIB programming and instrument control. These function statements, combined with SCPI, provide management and data communication for the GPIB interface and the signal generator.

This section describes functions used by different IO libraries. For more information, refer to the NI-488.2 Function Reference Manual for Windows, Agilent Standard Instrument Control Library reference manual, and Microsoft Visual C++ 6.0 documentation.

Abort Function

The HP Basic function `ABORT` and the other listed IO library functions terminate listener/talker activity on the GPIB and prepare the signal generator to receive a new command from the computer. Typically, this is an initialization command used to place the GPIB in a known starting condition.

Library	Function Statement	Initialization Command
HP Basic	The `ABORT` function stops all GPIB activity.	`10 ABORT 7`
VISA Library	In VISA, the viTerminate command requests a VISA session to terminate normal execution of an asynchronous operation. The parameter list describes the session and job id.	`viTerminate` (parameter list)
NI-488.2	The NI-488.2 library function aborts any asynchronous read, write, or command operation that is in progress. The parameter `ud` is the interface or device descriptor.	`ibstop(int ud)`

Library	Function Statement	Initialization Command
SICL	The Agilent SICL function aborts any command currently executing with the session `id`. This function is supported with C/C++ on Windows 3.1 and Series 700 HP-UX.	`iabort (id)`

Remote Function

The HP Basic function `REMOTE` and the other listed IO library functions change the signal generator from local operation to remote operation. In remote operation, the front panel keys are disabled except for the **Local** key and the line power switch. Pressing the **Local** key restores manual operation.

Library	Function Statement	Initialization Command
HP Basic	The `REMOTE 719` function disables the front panel operation of all keys with the exception of the **Local** key.	10 REMOTE 719
VISA Library	The VISA library, at this time, does not have a similar command.	N/A
NI-488.2	The NI-488.2 library function asserts the Remote Enable (REN) GPIB line. All devices listed in the parameter list are put into a listen-active state although no indication is generated by the signal generator. The parameter list describes the interface or device descriptor.	`EnableRemote` (parameter list)
SICL	The Agilent SICL function puts an instrument, identified by the `id` parameter, into remote mode and disables the front panel keys. Pressing the **Local** key on the signal generator front panel restores manual operation. The parameter id is the session identifier.	`iremote` (id)

Local Lockout Function

The HP Basic function `LOCAL LOCKOUT` and the other listed IO library functions disable the front panel keys including the **Local** key. With the **Local** key disabled, only the controller (or a hard reset of line power) can restore local control.

Library	Function Statement	Initialization Command
HP Basic	The `LOCAL LOCKOUT` function disables all front-panel signal generator keys. Return to local control can occur only by cycling power on the instrument, when the `LOCAL` command is sent or if the **Preset** key is pressed.	10 LOCAL LOCKOUT 719
VISA Library	The VISA library, at this time, does not have a similar command.	N/A

Library	Function Statement	Initialization Command
NI-488.2	The LOCAL LOCKOUT function disables all front-panel signal generator keys. Return to local control can occur only by cycling power on the instrument, when the LOCAL command is sent or if the **Preset** key is pressed.	SetRWLS (parameter list)
SICL	The Agilent SICL igpibllo prevents function prevents user access to front panel keys operation. The function puts an instrument, identified by the id parameter, into remote mode with local lockout. The parameter id is the session identifier and instrument address list.	igpibllo (id)

Local Function

The HP Basic function LOCAL and the other listed functions return the signal generator to local control with a fully enabled front panel.

Library	Function Statement	Initialization Command
HP Basic	The LOCAL 719 function returns the signal generator to manual operation, allowing access to the signal generator's front panel keys.	10 LOCAL 719
VISA Library	The VISA library, at this time, does not have a similar command.	N/A
NI-488.2	The NI-488.2 library function places the interface in local mode and allows operation of the signal generator's front panel keys. The ud parameter in the parameter list is the interface or device descriptor.	ibloc (int ud)
SICL	The Agilent SICL function puts the signal generator into Local operation; enabling front panel key operation. The id parameter identifies the session.	iloc(id)

Clear Function

The HP Basic function CLEAR and the other listed IO library functions clear the signal generator.

Library	Function Statement	Initialization Command
HP Basic	The CLEAR 719 function halts all pending output-parameter operations, resets the parser (interpreter of programming codes) and prepares for a new programming code, stops any sweep in progress, and turns off continuous sweep.	10 CLEAR 719
VISA Library	The VISA library uses the viClear function. This function performs an IEEE 488.1 clear of the signal generator.	viClear(ViSession vi)

Library	Function Statement	Initialization Command
NI-488.2	The NI-488.2 library function sends the GPIB Selected Device Clear (SDC) message to the device described by `ud`.	`ibclr(int ud)`
SICL	The Agilent SICL function clears a device or interface. The function also discards data in both the read and write formatted IO buffers. The `id` parameter identifies the session.	`iclear (id)`

Output Function

The HP Basic IO function `OUTPUT` and the other listed IO library functions put the signal generator into a listen mode and prepare it to receive ASCII data, typically SCPI commands.

Library	Function Statement	Initialization Command
HP Basic	The function `OUTPUT 719` puts the signal generator into remote mode, makes it a listener, and prepares it to receive data.	`10 OUTPUT 719`
VISA Library	The VISA library uses the above function and associated parameter list to output data. This function formats according to the format string and sends data to the device. The parameter list describes the session id and data to send.	viPrintf(parameter list)
NI-488.2	The NI-488.2 library function addresses the GPIB and writes data to the signal generator. The parameter list includes the instrument address, session id, and the data to send.	`ibwrt(parameter list)`
SICL	The Agilent SICL function converts data using the format string. The format string specifies how the argument is converted before it is output. The function sends the characters in the format string directly to the instrument. The parameter list includes the instrument address, data buffer to write, and so forth.	`iprintf (parameter list)`

Enter Function

The HP Basic function `ENTER` reads formatted data from the signal generator. Other IO libraries use similar functions to read data from the signal generator.

Library	Function Statement	Initialization Command
HP Basic	The function `ENTER 719` puts the signal generator into remote mode, makes it a talker, and assigns data or status information to a designated variable.	10 ENTER 719;
VISA Library	The VISA library uses the viScanf function and an associated parameter list to receive data. This function receives data from the instrument, formats it using the format string, and stores the data in the argument list. The parameter list includes the session id and string argument.	viScanf (parameter list)
NI-488.2	The NI-488.2 library function addresses the GPIB, reads data bytes from the signal generator, and stores the data into a specified buffer. The parameter list includes the instrument address and session id.	ibrd (parameter list)
SICL	The Agilent SICL function reads formatted data, converts it, and stores the results into the argument list. The conversion is done using conversion rules for the format string. The parameter list includes the instrument address, formatted data to read, and so forth.	iscanf (parameter list)

Interface Check using HP Basic and GPIB

This simple program causes the signal generator to perform an instrument reset. The SCPI command `*RST` places the signal generator into a pre-defined state and the remote annunciator (`R`) appears on the front panel display.

The following program example is available on the signal generator Documentation CD-ROM as `basicex1.txt`.

```
10    !*****************************************************************************
20    !
30    !  PROGRAM NAME:         basicex1.txt
40    !
50    !  PROGRAM DESCRIPTION:  This program verifies that the GPIB connections and
60    !                        interface are functional.
70    !
80    !  Connect a controller to the signal generator using a GPIB cable.
90    !
100   !
110   !  CLEAR and RESET the controller and type in the following commands and then
120   !  RUN the program:
130   !
```

```
140   !*******************************************************************************
150   !
160   Sig_gen=719    ! Declares a variable to hold the signal generator's address
170   LOCAL Sig_gen  ! Places the signal generator into Local mode
180   CLEAR Sig_gen  ! Clears any pending data I/O and resets the parser
190   REMOTE 719     ! Puts the signal generator into remote mode
200   CLEAR SCREEN   ! Clears the controllers display
210   REMOTE 719
220   OUTPUT Sig_gen;"*RST"  ! Places the signal generator into a defined state
230   PRINT "The signal generator should now be in REMOTE."
240   PRINT
250   PRINT "Verify that the remote [R] annunciator is on.  Press the `Local' key, "
260   PRINT "on the front panel to return the signal generator to local control."
270   PRINT
280   PRINT "Press RUN to start again."
290   END   ! Program ends
```

Interface Check Using NI-488.2 and C++

This example uses the NI-488.2 library to verify that the GPIB connections and interface are functional. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file.

The following program example is available on the signal generator Documentation CD-ROM as `niex1.cpp`.

```
// ******************************************************************************************
//
// PROGRAM NAME: niex1.cpp
//
// PROGRAM DESCRIPTION: This program verifies that the GPIB connections and
// interface are functional.
//
// Connect a GPIB cable from the PC GPIB card to the signal generator
// Enter the following code into the source .cpp file and execute the program
//
// ******************************************************************************************

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;

int GPIB0=   0;         // Board handle
```

```
Addr4882_t Address[31]; // Declares an array of type Addr4882_t

int main(void)

{
     int sig;                          // Declares a device descriptor variable
     sig = ibdev(0, 19, 0, 13, 1, 0); // Aquires a device descriptor
     ibclr(sig);                       // Sends device clear message to signal generator
     ibwrt(sig, "*RST", 4);            // Places the signal generator into a defined state

                                       // Print data to the output window
     cout << "The signal generator should now be in REMOTE. The remote indicator"<<endl;
     cout <<"annunciator R should appear on the signal generator display"<<endl;

  return 0;

}
```

Interface Check for GPIB Using VISA and C

This program uses VISA library functions and the C language to communicate with the signal generator. The program verifies that the GPIB connections and interface are functional. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex1.cpp` performs the following functions:

- verifies the GPIB connections and interface are functional
- switches the signal generator into remote operation mode

The following program example is available on the signal generator Documentation CD-ROM as `visaex1.cpp`.

```
//*********************************************************************************************
// PROGRAM NAME:visaex1.cpp
//
// PROGRAM DESCRIPTION:This example program verifies that the GPIB connections and
// and interface are functional.
// Turn signal generator power off then on and then run the program
//
//*********************************************************************************************

#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>

void main ()
```

```
{
ViSession defaultRM, vi;        // Declares a variable of type ViSession
                                // for instrument communication
ViStatus viStatus = 0;
                                      // Opens a session to the GPIB device
                                      // at address 19
viStatus=viOpenDefaultRM(&defaultRM);
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

viPrintf(vi, "*RST\n");         // initializes signal generator
                                // prints to the output window
printf("The signal generator should now be in REMOTE. The remote                    indicator\n");
printf("annunciator R should appear on the signal generator display\n");
printf("\n");

viClose(vi);                    // closes session
viClose(defaultRM);             // closes default session
}
```

Local Lockout Using HP Basic and GPIB

This example demonstrates the Local Lockout function. Local Lockout disables the front panel signal generator keys. `basicex2.txt` performs the following functions:

- resets instrument
- places signal generator into local
- places signal generator into remote

The following program example is available on the signal generator Documentation CD-ROM as `basicex2.txt`.

```
10    !************************************************************************
20    !
30    !  PROGRAM NAME:         basicex2.txt
40    !
50    !  PROGRAM DESCRIPTION:  In REMOTE mode, access to the signal generators
60    !                        functional front panel keys are disabled except for
70    !                        the Local and Contrast keys.  The LOCAL LOCKOUT
80    !                        command will disable the Local key.
90    !                        The LOCAL command, executed from the controller, is then
100   !                        the only way to return the signal generator to front panel,
```

```
110   !                        Local, control.
120   !*************************************************************************
130   Sig_gen=719       ! Declares a variable to hold signal generator address
140   CLEAR Sig_gen     ! Resets signal generator parser and clears any output
150   LOCAL Sig_gen     ! Places the signal generator in local mode
160   REMOTE Sig_gen    ! Places the signal generator in remote mode
170   CLEAR SCREEN      ! Clears the controllers display
180   OUTPUT Sig_gen;"*RST"       ! Places the signal generator in a defined state
190   ! The following print statements are user prompts
200   PRINT "The signal generator should now be in remote."
210   PRINT "Verify that the 'R' and 'L' annunciators are visable"
220   PRINT ".......... Press Continue"
230   PAUSE
240   LOCAL LOCKOUT 7   ! Puts the signal generator in LOCAL LOCKOUT mode
250   PRINT             ! Prints user prompt messages
260   PRINT "Signal generator should now be in LOCAL LOCKOUT mode."
270   PRINT
280   PRINT "Verify that all keys including `Local' (except Contrast keys) have no effect."
290   PRINT
300   PRINT ".......... Press Continue"
310   PAUSE
320   PRINT
330   LOCAL 7           ! Returns signal generator to Local control
340   ! The following print statements are user prompts
350   PRINT "Signal generator should now be in Local mode."
360   PRINT
370   PRINT "Verify that the signal generator's front-panel keyboard is functional."
380   PRINT
390   PRINT "To re-start this program press RUN."
400   END
```

Local Lockout Using NI-488.2 and C++

This example uses the NI-488.2 library to set the signal generator local lockout mode. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `niex2.cpp` performs the following functions:

- all front panel keys, except the contrast key
- places the signal generator into remote
- prompts the user to verify the signal generator is in remote
- places the signal generator into local

The following program example is available on the signal generator Documentation CD-ROM as `niex2.cpp`.

```
// ****************************************************************************
```

```
// PROGRAM NAME: niex2.cpp
//
// PROGRAM DESCRIPTION: This program will place the signal generator into
// LOCAL LOCKOUT mode. All front panel keys, except the Contrast key, will be disabled.
// The local command, 'ibloc(sig)' executed via program code, is the only way to
// return the signal generator to front panel, Local, control.
// ********************************************************************************

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;
int GPIB0=   0;                          // Board handle
Addr4882_t Address[31];                  // Declares a variable of type Addr4882_t

int main()

{
      int sig;                           // Declares variable to hold interface descriptor
      sig = ibdev(0, 19, 0, 13, 1, 0);   // Opens and initialize a device descriptor
      ibclr(sig);                        // Sends GPIB Selected Device Clear (SDC) message
      ibwrt(sig, "*RST", 4);             // Places signal generator in a defined state
      cout << "The signal generator should now be in REMOTE. The remote mode R "<<endl;
      cout <<"annunciator should appear on the signal generator display."<<endl;
      cout <<"Press Enter to continue"<<endl;
      cin.ignore(10000,'\n');
      SendIFC(GPIB0);                    // Resets the GPIB interface
      Address[0]=19;                     // Signal generator's address
      Address[1]=NOADDR;                 // Signifies end element in array. Defined in
                                         // DECL-32.H
      SetRWLS(GPIB0, Address);           // Places device in Remote with Lockout State.


      cout<< "The signal generator should now be in LOCAL LOCKOUT. Verify that all
             keys"<<endl;
      cout<< "including the 'Local' key are disabled (Contrast keys are not
              affected)"<<endl;
      cout <<"Press Enter to continue"<<endl;
      cin.ignore(10000,'\n');
      ibloc(sig);                        // Returns signal generator to local control
      cout<<endl;
      cout <<"The signal generator should now be in local mode\n";
 return 0;}
}
```

Queries Using HP Basic and GPIB

This example demonstrates signal generator query commands. The signal generator can be queried for conditions and setup parameters. Query commands are identified by the question mark as in the identify command `*IDN?` `basicex3.txt` performs the following functions:

- clears the signal generator
- queries the signal generator's settings

The following program example is available on the signal generator Documentation CD-ROM as `basicex3.txt`.

```
10    !*****************************************************************************
20    !
30    !  PROGRAM NAME:         basicex3.txt
40    !
50    !  PROGRAM DESCRIPTION:  In this example, query commands are used with response
60    !                        data formats.
70    !
80    !  CLEAR and RESET the controller and RUN the following program:
90    !
100   !*****************************************************************************
110   !
120   DIM A$[10],C$[100],D$[10]   ! Declares variables to hold string response data
130   INTEGER B                   ! Declares variable to hold integer response data
140   Sig_gen=719                 ! Declares variable to hold signal generator address
150   LOCAL Sig_gen               ! Puts signal generator in Local mode
160   CLEAR Sig_gen               ! Resets parser and clears any pending output
170   CLEAR SCREEN                ! Clears the controller's display
180   OUTPUT Sig_gen;"*RST"       ! Puts signal generator into a defined state
190   OUTPUT Sig_gen;"FREQ:CW?"   ! Querys the signal generator CW frequency setting
200   ENTER Sig_gen;F             ! Enter the CW frequency setting
210   ! Print frequency setting to the controller display
220   PRINT "Present source CW frequency is: ";F/1.E+6;"MHz"
230   PRINT
240   OUTPUT Sig_gen;"POW:AMPL?"  ! Querys the signal generator power level
250   ENTER Sig_gen;W             ! Enter the power level
260   ! Print power level to the controller display
270   PRINT "Current power setting is: ";W;"dBM"
280   PRINT
290   OUTPUT Sig_gen;"FREQ:MODE?" ! Querys the signal generator for frequency mode
300   ENTER Sig_gen;A$            ! Enter in the mode: CW, Fixed or List
310   ! Print frequency mode to the controller display
320   PRINT "Source's frequency mode is: ";A$
330   PRINT
```

```
340   OUTPUT Sig_gen;"OUTP OFF"   ! Turns signal generator RF state off
350   OUTPUT Sig_gen;"OUTP?"      ! Querys the operating state of the signal generator
360   ENTER Sig_gen;B             ! Enter in the state (0 for off)
370   ! Print the on/off state of the signal generator to the controller display
380   IF B>0 THEN
390     PRINT "Signal Generator output is: on"
400   ELSE
410     PRINT "Signal Generator output is: off"
420   END IF
430   OUTPUT Sig_gen;"*IDN?"      ! Querys for signal generator ID
440   ENTER Sig_gen;C$            ! Enter in the signal generator ID
450   ! Print the signal generator ID to the controller display
460   PRINT
470   PRINT "This signal generator is a ";C$
480   PRINT
490   ! The next command is a query for the signal generator's GPIB address
500   OUTPUT Sig_gen;"SYST:COMM:GPIB:ADDR?"
510   ENTER Sig_gen;D$            ! Enter in the signal generator's address
520   ! Print the signal generator's GPIB address to the controllers display
530   PRINT "The GPIB address is ";D$
540   PRINT
550   ! Print user prompts to the controller's display
560   PRINT "The signal generator is now under local control"
570   PRINT "or  Press RUN to start again."
580   END
```

Queries Using NI-488.2 and Visual C++

This example uses the NI-488.2 library to query different instrument states and conditions. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `niex3.cpp` performs the following functions:

- resets the signal generator
- queries the signal generator for various settings
- reads the various settings

The following program example is available on the signal generator Documentation CD-ROM as `niex3.cpp`.

```
//**************************************************************************************
// PROGRAM NAME: niex3.cpp
//
// PROGRAM DESCRIPTION: This example demonstrates the use of query commands.
//
// The signal generator can be queried for conditions and instrument states.
// These commands are of the type "*IDN?" where the question mark indicates
```

```
// a query.
//
//*************************************************************************************

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "Decl-32.h"
using namespace std;
int GPIB0=   0;                          // Board handle
Addr4882_t Address[31];                  // Declare a variable of type Addr4882_t

int main()

{
  int sig;                        // Declares variable to hold interface descriptor
  int num;
  char rdVal[100];                // Declares variable to read instrument responses
  sig = ibdev(0, 19, 0, 13, 1, 0); // Open and initialize a device descriptor
  ibloc(sig);                     // Places the signal generator in local mode
  ibclr(sig);                     // Sends Selected Device Clear(SDC) message
  ibwrt(sig, "*RST", 4);          // Places signal generator in a defined state
  ibwrt(sig, ":FREQuency:CW?",14); // Querys the CW frequency
  ibrd(sig, rdVal,100);           // Reads in the response into rdVal
  rdVal[ibcntl] = '\0';           // Null character indicating end of array
  cout<<"Source CW frequency is "<<rdVal;   // Print frequency of signal generator
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, "POW:AMPL?",10);     // Querys the signal generator
  ibrd(sig, rdVal,100);           // Reads the signal generator power level
  rdVal[ibcntl] = '\0';           // Null character indicating end of array
                                 // Prints signal generator power level
  cout<<"Source power (dBm) is : "<<rdVal;
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, ":FREQ:MODE?",11);    // Querys source frequency mode
  ibrd(sig, rdVal,100);           // Enters in the source frequency mode
  rdVal[ibcntl] = '\0';           // Null character indicating end of array
  cout<<"Source frequency mode is "<<rdVal; // Print source frequency mode
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, "OUTP OFF",12);       // Turns off RF source
```

```
  ibwrt(sig, "OUTP?",5);           // Querys the on/off state of the instrument
  ibrd(sig,rdVal,2);               // Enter in the source state
  rdVal[ibcntl] = '\0';
  num = (int (rdVal[0]) -('0'));
  if (num > 0){
      cout<<"Source RF state is : On"<<endl;
  }else{
      cout<<"Source RF state is : Off"<<endl;}
  cout<<endl;
  ibwrt(sig, "*IDN?",5);           // Querys the instrument ID
  ibrd(sig, rdVal,100);            // Reads the source ID
  rdVal[ibcntl] = '\0';            // Null character indicating end of array
  cout<<"Source ID is : "<<rdVal;  // Prints the source ID
  cout<<"Press any key to continue"<<endl;
  cin.ignore(10000,'\n');
  ibwrt(sig, "SYST:COMM:GPIB:ADDR?",20); //Querys source address
  ibrd(sig, rdVal,100);            // Reads the source address
  rdVal[ibcntl] = '\0';            // Null character indicates end of array
                                  // Prints the signal generator address
  cout<<"Source GPIB address is : "<<rdVal;
  cout<<endl;
  cout<<"Press the 'Local' key to return the signal generator to LOCAL control”<<endl;      cout<<endl;
return 0;
}
```

Queries for GPIB Using VISA and C

This example uses VISA library functions to query different instrument states and conditions. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex3.cpp` performs the following functions:

- verifies the GPIB connections and interface are functional
- resets the signal generator
- queries the instrument (CW frequency, power level, frequency mode, and RF state)
- reads responses into the rdBuffer (CW frequency, power level, and frequency mode)
- turns signal generator RF state off
- verifies RF state off

The following program example is available on the signal generator Documentation CD-ROM as `visaex3.cpp`.

```
//*************************************************************************************
// PROGRAM FILE NAME:visaex3.cpp
//
// PROGRAM DESCRIPTION:This example demonstrates the use of query commands. The signal
// generator can be queried for conditions and instrument states. These commands are of
// the type "*IDN?"; the question mark indicates a query.
```

```
//
//**********************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <conio.h>
#include <stdlib.h>
using namespace std;


void main ()
{
ViSession defaultRM, vi;    // Declares variables of type ViSession
                            // for instrument communication
ViStatus viStatus = 0;      // Declares a variable of type ViStatus
                            // for GPIB verifications
char rdBuffer [256];        // Declares variable to hold string data
int num;                    // Declares variable to hold integer data
                            // Initialize the VISA system
viStatus=viOpenDefaultRM(&defaultRM);
                            // Open session to GPIB device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){               // If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}
viPrintf(vi, "*RST\n");         // Resets signal generator
viPrintf(vi, "FREQ:CW?\n");     // Querys the CW frequency
viScanf(vi, "%t", rdBuffer);    // Reads response into rdBuffer
                                // Prints the source frequency
printf("Source CW frequency is : %s\n", rdBuffer);
printf("Press any key to continue\n");
printf("\n");                   // Prints new line character to the display
getch();
viPrintf(vi, "POW:AMPL?\n");    // Querys the power level
viScanf(vi, "%t", rdBuffer);    // Reads the response into rdBuffer
                                // Prints the source power level
printf("Source power (dBm) is : %s\n", rdBuffer);
printf("Press any key to continue\n");
printf("\n");                   // Prints new line character to the display
```

```
getch();
viPrintf(vi, "FREQ:MODE?\n");   // Querys the frequency mode
viScanf(vi, "%t", rdBuffer);    // Reads the response into rdBuffer
                                // Prints the source freq mode
printf("Source frequency mode is : %s\n", rdBuffer);
printf("Press any key to continue\n");
printf("\n");                   // Prints new line character to the display
getch();
viPrintf(vi, "OUTP OFF\n");     // Turns source RF state off
viPrintf(vi, "OUTP?\n");        // Querys the signal generator's RF state
viScanf(vi, "%1i", &num);       // Reads the response (integer value)
                                // Prints the on/off RF state
 if (num > 0 ) {
printf("Source RF state is : on\n");
}else{
printf("Source RF state is : off\n");
}
                                  // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Generating a CW Signal Using VISA and C

This example uses VISA library functions to control the signal generator. The signal generator is set for a CW frequency of 500 kHz and a power level of –2.3 dBm. Launch Microsoft Visual C++ 6.0, add the required files, and enter the code into your .cpp source file. `visaex4.cpp` performs the following functions:

- verifies the GPIB connections and interface are functional
- resets the signal generator
- queries the instrument (CW frequency, power level, frequency mode, and RF state)
- reads responses into the rdBuffer (CW frequency, power level, and frequency mode)
- turns signal generator RF state off
- verifies RF state off

The following program example is available on the signal generator Documentation CD-ROM as `visaex4.cpp`.

```
//*****************************************************************************
// PROGRAM FILE NAME:   visaex4.cpp
//
// PROGRAM DESCRIPTION: This example demonstrates query commands. The signal generator
// frequency and power level.
// The RF state of the signal generator is turn on and then the state is queried. The
// response will indicate that the RF state is on. The RF state is then turned off and
// queried. The response should indicate that the RF state is off. The query results are
```

```
// printed to the to the display window.
//
//*********************************************************************************************

#include "StdAfx.h"
#include <visa.h>
#include <iostream>
#include <stdlib.h>
#include <conio.h>

void main ()
{
 ViSession   defaultRM, vi;      // Declares variables of type ViSession
                                 // for instrument communication
ViStatus viStatus = 0;           // Declares a variable of type ViStatus
                                 // for GPIB verifications
char rdBuffer [256];             // Declare variable to hold string data
int num;                         // Declare variable to hold integer data

viStatus=viOpenDefaultRM(&defaultRM);     // Initialize VISA system
                                 // Open session to GPIB device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                    // If problems then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

viPrintf(vi, "*RST\n");          // Reset the signal generator
viPrintf(vi, "FREQ 500 kHz\n"); // Set the source CW frequency for 500 kHz
viPrintf(vi, "FREQ:CW?\n");      // Query the CW frequency
viScanf(vi, "%t", rdBuffer);     // Read signal generator response
printf("Source CW frequency is : %s\n", rdBuffer);  // Print the frequency
viPrintf(vi, "POW:AMPL -2.3 dBm\n");  // Set the power level to -2.3 dBm
viPrintf(vi, "POW:AMPL?\n");     // Query the power level
viScanf(vi, "%t", rdBuffer);     // Read the response into rdBuffer
printf("Source power (dBm) is : %s\n", rdBuffer); // Print the power level
viPrintf(vi, "OUTP:STAT ON\n"); // Turn source RF state on
viPrintf(vi, "OUTP?\n");         // Query the signal generator's RF state
viScanf(vi, "%1i", &num);        // Read the response (integer value)
  // Print the on/off RF state
if (num > 0 ) {
```

```
printf("Source RF state is : on\n");
}else{
printf("Source RF state is : off\n");
}
printf("\n");
printf("Verify RF state then press continue\n");
printf("\n");
getch();
viClear(vi);
viPrintf(vi,"OUTP:STAT OFF\n"); // Turn source RF state off
viPrintf(vi, "OUTP?\n");        // Query the signal generator's RF state
viScanf(vi, "%1i", &num);       // Read the response
  // Print the on/off RF state
 if (num > 0 ) {
printf("Source RF state is now: on\n");
}else{
printf("Source RF state is now: off\n");
}
                                    // Close the sessions
printf("\n");
viClear(vi);
viClose(vi);
viClose(defaultRM);
}
```

Generating an Externally Applied AC-Coupled FM Signal Using VISA and C

In this example, the VISA library is used to generate an ac-coupled FM signal at a carrier frequency of 700 MHz, a power level of –2.5 dBm, and a deviation of 20 kHz. Before running the program:

- Connect the output of a modulating signal source to the signal generator's EXT 2 input connector.
- Set the modulation signal source for the desired FM characteristics.

Launch Microsoft Visual C++ 6.0, add the required files, and enter the code into your .cpp source file. `visaex5.cpp` performs the following functions:

- error checking
- resets the signal generator
- sets up the EXT 2 connector on the signal generator for FM
- sets up FM path 2 coupling to AC
- sets up FM path 2 deviation to 20 kHz
- sets carrier frequency to 700 MHz
- sets the power level to -2.5 dBm
- turns on frequency modulation and RF output

The following program example is available on the signal generator Documentation CD-ROM as `visaex5.cpp`.

```
//*****************************************************************************************
```

```
// PROGRAM FILE NAME:visaex5.cpp
//
// PROGRAM DESCRIPTION:This example sets the signal generator FM source to External 2,
// coupling to AC, deviation to 20 kHZ, carrier frequency to 700 MHz and the power level
// to -2.5 dBm. The RF state is set to on.
//
//*****************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>

void main ()
{
 ViSession defaultRM, vi;           // Declares variables of type ViSession
                                    // for instrument communication
ViStatus viStatus = 0;              // Declares a variable of type ViStatus
                                    // for GPIB verifications
                                    // Initialize VISA session
viStatus=viOpenDefaultRM(&defaultRM);
                                // open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                       // If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}

printf("Example program to set up the signal generator\n");
printf("for an AC-coupled FM signal\n");
printf("Press any key to continue\n");
printf("\n");
getch();
printf("\n");

viPrintf(vi, "*RST\n");             // Resets the signal generator
viPrintf(vi, "FM:SOUR EXT2\n");     // Sets EXT 2 source for FM
viPrintf(vi, "FM:EXT2:COUP AC\n");  // Sets FM path 2 coupling to AC
viPrintf(vi, "FM:DEV 20 kHz\n");    // Sets FM path 2 deviation to 20 kHz
viPrintf(vi, "FREQ 700 MHz\n");     // Sets carrier frequency to 700 MHz
```

```
viPrintf(vi, "POW:AMPL -2.5 dBm\n"); // Sets the power level to -2.5 dBm
viPrintf(vi, "FM:STAT ON\n");        // Turns on frequency modulation
viPrintf(vi, "OUTP:STAT ON\n");      // Turns on RF output
                                     // Print user information
printf("Power level : -2.5 dBm\n");
printf("FM state : on\n");
printf("RF output : on\n");
printf("Carrier Frequency : 700 MHZ\n");
printf("Deviation : 20 kHZ\n");
printf("EXT2 and AC coupling are selected\n");
printf("\n");                        // Prints a carrage return
                                     // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Generating an Internal FM Signal Using VISA and C

In this example the VISA library is used to generate an internal FM signal at a carrier frequency of 900 MHz and a power level of –15 dBm. The FM rate will be 5 kHz and the peak deviation will be 100 kHz. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex6.cpp` performs the following functions:

- error checking
- resets the signal generator
- sets up the signal generator for FM path 2 and internal FM rate of 5 kHz
- sets up FM path 2 deviation to 100 kHz
- sets carrier frequency to 900 MHz
- sets the power level to -15 dBm
- turns on frequency modulation and RF output

The following program example is available on the signal generator Documentation CD-ROM as `visaex6.cpp`.

```
//*****************************************************************************************
// PROGRAM FILE NAME:visaex6.cpp
//
// PROGRAM DESCRIPION:This example generates an internal FM signal at a 900
// MHz carrier frequency and a power level of -15 dBm. The FM rate is 5 kHz and the peak
// deviation 100 kHz
//
//*****************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <stdlib.h>
```

```
#include <conio.h>

void main ()
{
ViSession defaultRM, vi;           // Declares variables of type ViSession
                                   // for instrument communication
ViStatus viStatus = 0;             // Declares a variable of type ViStatus
                                   // for GPIB verifications

viStatus=viOpenDefaultRM(&defaultRM); // Initialize VISA session
                              // open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                         // If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}

printf("Example program to set up the signal generator\n");
printf("for an AC-coupled FM signal\n");
printf("\n");
printf("Press any key to continue\n");
getch();
viClear(vi);                          // Clears the signal generator
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "FM2:INT:FREQ 5 kHz\n"); // Sets FM path 2 to internal at a modulation rate of 5 kHz
viPrintf(vi, "FM2:DEV 100 kHz\n");    // Sets FM path 2 modulation deviation rate of 100 kHz
viPrintf(vi, "FREQ 900 MHz\n");       // Sets carrier frequency to 900 MHz
viPrintf(vi, "POW -15 dBm\n");        // Sets the power level to -15 dBm
viPrintf(vi, "FM2:STAT ON\n");        // Turns on frequency modulation
viPrintf(vi, "OUTP:STAT ON\n");       // Turns on RF output
printf("\n");                         // Prints a carriage return
                                       // Print user information
printf("Power level : -15 dBm\n");
printf("FM state : on\n");
printf("RF output : on\n");
printf("Carrier Frequency : 900 MHZ\n");
printf("Deviation : 100 kHZ\n");
printf("Internal modulation : 5 kHz\n");
printf("\n");                        // Print a carrage return
                    // Close the sessions
viClose(vi);
```

```
viClose(defaultRM);
}
```

Generating a Step-Swept Signal Using VISA and C++

In this example the VISA library is used to set the signal generator for a continuous step sweep on a defined set of points from 500 MHz to 800 MHz. The number of steps is set for 10 and the dwell time at each step is set to 500 ms. The signal generator will then be set to local mode which allows the user to make adjustments from the front panel. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex7.cpp` performs the following functions:

- clears and resets the signal generator
- sets up the instrument for continuous step sweep
- sets up the start and stop sweep frequencies
- sets up the number of steps
- sets the power level
- turns on the RF output

The following program example is available on the signal generator Documentation CD-ROM as `visaex7.cpp`.

```
//*************************************************************************************
// PROGRAM FILE NAME:visaex7.cpp
//
// PROGRAM DESCRIPTION:This example will program the signal generator to perform a step
// sweep from 500-800 MHz with a .5 sec dwell at each frequency step.
//
//*************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>

void main ()
{
ViSession defaultRM, vi;// Declares variables of type ViSession
// vi establishes instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
                        // for GPIB verifications

viStatus=viOpenDefaultRM(&defaultRM); // Initialize VISA session
             // Open session to GPIB device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){// If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
```

```
printf("\n");
exit(0);}

viClear(vi);                          // Clears the signal generator
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "*CLS\n");               // Clears the status byte register
viPrintf(vi, "FREQ:MODE LIST\n");     // Sets the sig gen freq mode to list
viPrintf(vi, "LIST:TYPE STEP\n");     // Sets sig gen LIST type to step
viPrintf(vi, "FREQ:STAR 500 MHz\n");  // Sets start frequency
viPrintf(vi, "FREQ:STOP 800 MHz\n");  // Sets stop frequency
viPrintf(vi, "SWE:POIN 10\n");        // Sets number of steps (30 mHz/step)
viPrintf(vi, "SWE:DWEL .5 S\n");      // Sets dwell time to 500 ms/step
viPrintf(vi, "POW:AMPL -5 dBm\n");    // Sets the power level for -5 dBm
viPrintf(vi, "OUTP:STAT ON\n");       // Turns RF output on
viPrintf(vi, "INIT:CONT ON\n");       // Begins the step sweep operation
                                      // Print user information
printf("The signal generator is in step sweep mode. The frequency range                          is\n");
printf("500 to 800 mHz. There is a .5 sec dwell time at each 30 mHz                            step.\n");
printf("\n");                         // Prints a carriage return/line feed
 viPrintf(vi, "OUTP:STAT OFF\n");      // Turns the RF output off
printf("Press the front panel Local key to return the\n");
printf("signal generator to manual operation.\n");
                                      // Closes the sessions
printf("\n");
viClose(vi);
viClose(defaultRM);
}
```

Generating a Swept Signal Using VISA and Visual C++

This example sets up the signal generator for a frequency sweep from 1 to 2 GHz with 101 points and a .01 second dwell period for each point. A loop is used to generator 5 sweep operations. The signal generator triggers each sweep with the `:INIT` command. There is a wait introduced in the loop to allow the signal generator to complete all operations such as set up and retrace before the next sweep is generated. `visaex11.cpp` performs the following functions:

- sets up the signal generator for a 1 to 2 GHz frequency sweep
- sets up the signal generator to have a dwell time of .01 seconds and 101 points in the sweep
- sleep function is used to allow the instrument to complete its sweep operation

The following program example is available on the signal generator Documentation CD-ROM as `visaex11.cpp`.

```
//**********************************************************************
// PROGRAM FILE NAME: visaex11.cpp
//
// PROGRAM DESCRIPTION: This program sets up the signal generator to
```

```
// sweep from 1-2 GHz. A loop and counter are used to generate 5 sweeps.
// Each sweep consists of 101 points with a .01 second dwell at each point.
//
// The program uses a Sleep function to allow the signal generator to
// complete it's sweep operation before the INIT command is sent.
// The Sleep function is available with the windows.h header file which is
// included in the project.
//
// NOTE: Change the TCPIP0 address in the instOpenString declaration to
// match the IP address of your signal generator.
//
//*************************************************************************

#include "stdafx.h"
#include "visa.h"
#include <iostream>
#include <windows.h>

void main ()
 {
     ViStatus stat;
     ViSession defaultRM,inst;

     int npoints = 101;
     double dwell = 0.01;
     int intCounter=5;

     char* instOpenString = "TCPIP0::141.121.93.101::INSTR";

     stat = viOpenDefaultRM(&defaultRM);
     stat = viOpen(defaultRM,instOpenString,VI_NULL,VI_NULL, &inst);
     // preset to start clean

     stat = viPrintf( inst, "*RST\n" );
     // set power level for -10dBm
     stat = viPrintf(inst, "POW -10DBM\n");
     // set the start and stop frequency for the sweep
     stat = viPrintf(inst, "FREQ:START 1GHZ\n");
     stat = viPrintf(inst, "FREQ:STOP 2GHZ\n");
     // setup dwell per point
     stat = viPrintf(inst, "SWEEP:DWELL %e\n", dwell);
     // setup number of points
```

```
    stat = viPrintf(inst, "SWEEP:POINTS %d\n", npoints);

    // set interface timeout to double the expected sweep time
    // sweep takes (~15ms + dwell) per point * number of points
    // the timeout should not be shorter then the sweep, set it
    // longer
    long timeoutMS = long(2*npoints*(.015+dwell)*1000);
    // set the VISA timeout
    stat = viSetAttribute(inst, VI_ATTR_TMO_VALUE, timeoutMS);

    // set continuous trigger mode off
    stat = viPrintf(inst, "INIT:CONT OFF\n");
    // turn list sweep on
    stat = viPrintf(inst, "FREQ:MODE LIST\n");

    int sweepNo = 0;
    while(intCounter>0 )
    {
        // start the sweep (initialize)
        stat = viPrintf(inst, "INIT\n");
        printf("Sweep %d started\n",++sweepNo);
        // wait for the sweep completion with *OPC?
        int res ;
        stat = viPrintf(inst, "*OPC?\n");
        stat = viScanf(inst, "%d", &res);
        // handle possible errors here (most likely a timeout)
        // err_handler( inst, stat );
        puts("Sweep ended");
        // delay before sending next INIT since instrument
        // may not be ready to receive it yet
        Sleep(15);

intCounter = intCounter-1;

    }
printf("End of Program\n\n");

}
```

Saving and Recalling States Using VISA and C

In this example, instrument settings are saved in the signal generator's save register. These settings can then be recalled separately; either from the keyboard or from the signal generator's front panel.

Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex8.cpp` performs the following functions:

- error checking
- clears the signal generator
- resets the status byte register
- resets the signal generator
- sets up the signal generator frequency, ALC off, power level, RF output on
- checks for operation complete
- saves to settings to instrument register number one
- recalls information from register number one
- prompts user input to put instrument into Local and checks for operation complete

The following program example is available on the signal generator Documentation CD-ROM as `visaex8.cpp`.

```
//*************************************************************************************
// PROGRAM FILE NAME:visaex8.cpp
//
// PROGRAM DESCRIPTION:In this example, instrument settings are saved in the signal
// generator's registers and then recalled.
// Instrument settings can be recalled from the keyboard or, when the signal generator
// is put into Local control, from the front panel.
// This program will initialize the signal generator for an instrument state, store the
// state to register #1. An *RST command will reset the signal generator and a *RCL
// command will return it to the stored state. Following this remote operation the user
// will be instructed to place the signal generator in Local mode.
//
//*************************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <conio.h>

void main ()
{
 ViSession defaultRM, vi;// Declares variables of type ViSession
// for instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
                        // for GPIB verifications
long lngDone = 0;         // Operation complete flag

viStatus=viOpenDefaultRM(&defaultRM);     // Initialize VISA session
// Open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
```

```
if(viStatus){// If problems, then prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}
printf("\n");
viClear(vi);                          // Clears the signal generator
viPrintf(vi, "*CLS\n");               // Resets the status byte register
                                      // Print user information
printf("Programming example using the  *SAV,*RCL   SCPI commands\n");
printf("used to save and recall an instrument's state\n");
printf("\n");
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "FREQ 5 MHz\n");         // Sets sig gen frequency
viPrintf(vi, "POW:ALC OFF\n");        // Turns ALC Off
viPrintf(vi, "POW:AMPL -3.2 dBm\n");  // Sets power for -3.2 dBm
viPrintf(vi, "OUTP:STAT ON\n");       // Turns RF output On
viPrintf(vi, "*OPC?\n");              // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);      // Waits for setup to complete
viPrintf(vi, "*SAV 1\n");             // Saves sig gen state to register #1
                                      // Print user information
printf("The current signal generator operating state will be saved\n");
printf("to Register #1. Observe the state then press Enter\n");
printf("\n");                         // Prints new line character
getch();                              // Wait for user input
lngDone=0;                            // Resets the operation complete flag
viPrintf(vi, "*RST\n");               // Resets the signal generator
viPrintf(vi, "*OPC?\n");              // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);      // Waits for setup to complete
                                      // Print user infromation
printf("The instrument is now in it's Reset operating state. Press the\n");
printf("Enter key to return the signal generator to the Register #1                          state\n");
printf("\n");                         // Prints new line character
getch();                              // Waits for user input
lngDone=0;                            // Reset the operation complete flag
viPrintf(vi, "*RCL 1\n");             // Recalls stored register #1 state
viPrintf(vi, "*OPC?\n");              // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);      // Waits for setup to complete
                                      // Print user information
```

```
printf("The signal generator has been returned to it's Register #1                    state\n");
printf("Press Enter to continue\n");
printf("\n");                          // Prints new line character
getch();                               // Waits for user input
lngDone=0;                             // Reset the operation complete flag
viPrintf(vi, "*RST\n");                // Resets the signal generator
viPrintf(vi, "*OPC?\n");               // Checks for operation complete
while (!lngDone)
    viScanf (vi ,"%d",&lngDone);       // Waits for setup to complete
                                       // Print user information
printf("Press Local on instrument front panel to return to manual mode\n");
printf("\n");                          // Prints new line character
                                       // Close the sessions
viClose(vi);
viClose(defaultRM);
}
```

Reading the Data Questionable Status Register Using VISA and C

In this example, the signal generator's data questionable status register is read. You will be asked to set up the signal generator for error generating conditions. The data questionable status register will be read and the program will notify the user of the error condition that the setup caused. Follow the user prompts presented when the program runs. Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex9.cpp` performs the following functions:

- error checking
- clears the signal generator
- resets the signal generator
- the data questionable status register is enabled to read an unleveled condition
- prompts user to manually set up the signal generator for an unleveled condition
- queries the data questionable status register for any set bits and converts the string data to numeric
- based on the numeric value, program checks for a corresponding status check value
- similarly checks for over or undermodulation condition

The following program example is available on the signal generator Documentation CD-ROM as `visaex9.cpp`.

```
//*************************************************************************************
// PROGRAM NAME:visaex9.cpp
//
// PROGRAM DESCRIPTION:In this example, the data questionable status register is read.
// The data questionable status register is enabled to read an unleveled condition.
// The signal generator is then set up for an unleveled condition and the data
// questionable status register read. The results are then displayed to the user.
// The status questionable register is then setup to monitor a modulation error condition.
// The signal generator is set up for a modulation error condition and the data
```

```
// questionable status register is read.
// The results are displayed to the active window.
//
//*****************************************************************************

#include <visa.h>
#include "StdAfx.h"
#include <iostream>
#include <conio.h>

void main ()
{
ViSession defaultRM, vi;// Declares a variables of type ViSession
                        // for instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
// for GPIB verifications
int num=0;// Declares a variable for switch statements

char rdBuffer[256]={0};         // Declare a variable for response data

viStatus=viOpenDefaultRM(&defaultRM);     // Initialize VISA session
                             // Open session to GPIB device at address 19

viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                    // If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}
printf("\n");
viClear(vi);// Clears the signal generator
// Prints user information
printf("Programming example to demonstrate reading the signal generator's
                         Status Byte\n");
printf("\n");
printf("Manually set up the sig gen for an unleveled output condition:\n");
printf("* Set signal generator output amplitude to +20 dBm\n");
printf("* Set frequency to maximum value\n");
printf("* Turn On signal generator's RF Output\n");
printf("* Check signal generator's display for the UNLEVEL annunciator\n");
printf("\n");
printf("Press Enter when ready\n");
printf("\n");
```

```
getch();                                  // Waits for keyboard user input
viPrintf(vi, "STAT:QUES:POW:ENAB 2\n");   // Enables the Data Questionable
                                          // Power Condition Register Bits
         // Bits '0' and '1'
viPrintf(vi, "STAT:QUES:POW:COND?\n");    // Querys the register for any
         // set bits
viScanf(vi, "%s", rdBuffer);              // Reads the decimal sum of the
         // set bits
num=(int (rdBuffer[1]) -('0'));           // Converts string data to
         // numeric

switch (num)                              // Based on the decimal value
{
    case 1:
printf("Signal Generator Reverse Power Protection                                    Tripped\n");
printf("/n");
break;
    case 2:
printf("Signal Generator Power is Unleveled\n");
printf("\n");
break;
    default:
printf("No Power Unleveled condition detected\n");
printf("\n");
}
viClear(vi);                              // Clears the signal generator
                                          // Prints user information
printf("-----------------------------------------------------------------\n");
printf("\n");
printf("Manually set up the sig gen for an unleveled output condition:\n");
printf("\n");
printf("* Select AM modulation\n");
printf("* Select AM Source Ext 1 and Ext Coupling AC\n");
printf("* Turn On the modulation.\n");
printf("* Do not connect any source to the input\n");
printf("* Check signal generator's display for the EXT1 LO annunciator\n");
printf("\n");
printf("Press Enter when ready\n");
printf("\n");
getch();                                  // Waits for keyboard user input
viPrintf(vi, "STAT:QUES:MOD:ENAB 16\n");  // Enables the Data Questionable
                                          // Modulation Condition Register
```

```
        // bits '0','1','2','3' and  '4'
 viPrintf(vi, "STAT:QUES:MOD:COND?\n");    // Querys the register for any
        // set bits
  viScanf(vi, "%s", rdBuffer);              // Reads the decimal sum of the
        // set bits
num=(int (rdBuffer[1]) -('0')); // Converts string data to numeric

switch (num)                              // Based on the decimal value
{
    case 1:
printf("Signal Generator Modulation 1 Undermod\n");
printf("\n");
break;
    case 2:
printf("Signal Generator Modulation 1 Overmod\n");
printf("\n");
break;
    case 4:
printf("Signal Generator Modulation 2 Undermod\n");
printf("\n");
break;
    case 8:
printf("Signal Generator Modulation 2 Overmod\n");
printf("\n");
break;
    case 16:
printf("Signal Generator Modulation Uncalibrated\n");
printf("\n");
break;
    default:
printf("No Problems with Modulation\n");
printf("\n");
}
// Close the sessions
viClose(vi);
viClose(defaultRM);

}
```

Reading the Service Request Interrupt (SRQ) Using VISA and C

This example demonstrates use of the Service Request (SRQ) interrupt. By using the SRQ, the computer can attend to other tasks while the signal generator is busy performing a function or operation. When the signal generator finishes its operation, or detects a failure, then a Service Request can be generated. The computer will respond to the SRQ and, depending on the code, can perform some other operation or notify the user of failures or other conditions.

This program sets up a step sweep function for the signal generator and, while the operation is in progress, prints out a series of asterisks. When the step sweep operation is complete, an SRQ is generated and the printing ceases.

Launch Microsoft Visual C++ 6.0, add the required files, and enter the following code into your .cpp source file. `visaex10.cpp` performs the following functions:

- error checking
- clears the signal generator
- resets the signal generator
- prompts user to manually begin the step sweep and waits for response
- clears the status register
- sets up the operation status group to respond to an end of sweep
- the data questionable status register is enabled to read an unleveled condition
- prompts user to manually set up the signal generator for an unleveled condition
- queries the data questionable status register for any set bits and converts the string data to numeric
- based on the numeric value, program checks for a corresponding status check value
- similarly checks for over or undermodulation condition

The following program example is available on the signal generator Documentation CD-ROM as `visaex10.cpp`.

```
//**************************************************************************
//
// PROGRAM FILE NAME:visaex10.cpp
//
// PROGRAM DESCRIPTION: This example demonstrates the use of a Service Request (SRQ)
// interrupt. The program sets up conditions to enable the SRQ and then sets the signal
// generator for a step mode sweep. The program will enter a printing loop which prints
// an * character and ends when the sweep has completed and an SRQ received.
//
//**************************************************************************



#include "visa.h"
#include <stdio.h>
#include "StdAfx.h"
#include "windows.h"
#include <conio.h>

#define  MAX_CNT 1024
```

```
int sweep=1;  // End of sweep flag

/* Prototypes */

ViStatus _VI_FUNCH interupt(ViSession vi, ViEventType eventType, ViEvent event, ViAddr addr);

int main ()
{
ViSession defaultRM, vi;// Declares variables of type ViSession
// for instrument communication
ViStatus viStatus = 0;// Declares a variable of type ViStatus
                      // for GPIB verifications
char rdBuffer[MAX_CNT];// Declare a block of memory data

viStatus=viOpenDefaultRM(&defaultRM);// Initialize VISA session
if(viStatus < VI_SUCCESS){// If problems, then prompt user
printf("ERROR initializing VISA... exiting\n");
printf("\n");
return -1;}
                              // Open session to gpib device at address 19
viStatus=viOpen(defaultRM, "GPIB::19::INSTR", VI_NULL, VI_NULL, &vi);
if(viStatus){                 // If problems then prompt user
printf("ERROR: Could not open communication with
                                        instrument\n");
printf("\n");
return -1;}

viClear(vi);                  // Clears the signal generator
viPrintf(vi, "*RST\n");       // Resets signal generator
                              // Print program header and information
printf("** End of Sweep Service Request **\n");
printf("\n");
printf("The signal generator will be set up for a step sweep mode
      operation.\n");
printf("An '*' will be printed while the instrument is sweeping. The end of
 \n");
printf("sweep will be indicated by an SRQ on the GPIB and the program will
                        end.\n");
printf("\n");
printf("Press Enter to continue\n");
printf("\n");
getch();
```

```
viPrintf(vi, "*CLS\n");// Clears signal generator status byte

viPrintf(vi, "STAT:OPER:NTR 8\n");// Sets the Operation Status Group  // Negative Transition Filter to
indicate a  // negative transition in Bit 3 (Sweeping)

// which will set a corresponding event in   // the Operation Event Register. This occurs   // at the end
of a sweep.

viPrintf(vi, "STAT:OPER:PTR 0\n");// Sets the Operation Status Group  // Positive Transition Filter so
that no

// positive transition on Bit 3 affects the  // Operation Event Register. The positive  // transition
occurs at the start of a sweep.

viPrintf(vi, "STAT:OPER:ENAB 8\n");// Enables Operation Status Event Bit 3  // to report the event to
Status Byte  // Register Summary Bit 7.

viPrintf(vi, "*SRE 128\n");// Enables Status Byte Register Summary Bit 7

// The next line of code indicates the  // function to call on an event

viStatus = viInstallHandler(vi, VI_EVENT_SERVICE_REQ, interupt, rdBuffer);

// The next line of code enables the  // detection of an event

viStatus = viEnableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);


viPrintf(vi, "FREQ:MODE LIST\n");// Sets frequency mode to list

viPrintf(vi, "LIST:TYPE STEP\n");// Sets sweep to step

viPrintf(vi, "LIST:TRIG:SOUR IMM\n");// Immediately trigger the sweep

viPrintf(vi, "LIST:MODE AUTO\n");// Sets mode for the list sweep

viPrintf(vi, "FREQ:STAR 40 MHZ\n"); // Start frequency set to 40 MHz

viPrintf(vi, "FREQ:STOP 900 MHZ\n");// Stop frequency set to 900 MHz

viPrintf(vi, "SWE:POIN 25\n");// Set number of points for the step sweep

viPrintf(vi, "SWE:DWEL .5 S\n");// Allow .5 sec dwell at each point

viPrintf(vi, "INIT:CONT OFF\n");// Set up for single sweep

viPrintf(vi, "TRIG:SOUR IMM\n");// Triggers the sweep

viPrintf(vi, "INIT\n");  // Takes a single sweep

printf("\n");

// While the instrument is sweeping have the

// program busy with printing to the display.

// The Sleep function, defined in the header

// file windows.h, will pause the program

// operation for .5 seconds

while (sweep==1){

printf("*");

Sleep(500);}

printf("\n");

// The following lines of code will stop the

// events and close down the session


viStatus = viDisableEvent(vi, VI_ALL_ENABLED_EVENTS,VI_ALL_MECH);

viStatus = viUninstallHandler(vi, VI_EVENT_SERVICE_REQ, interupt,
                                                rdBuffer);

viStatus = viClose(vi);
```

```
viStatus = viClose(defaultRM);
return 0;

}

// The following function is called when an SRQ event occurs. Code specific to your
// requirements would be entered in the body of the function.

ViStatus _VI_FUNCH interupt(ViSession vi, ViEventType eventType, ViEvent event, ViAddr
                            addr)
{
ViStatus status;
ViUInt16 stb;

 status = viReadSTB(vi, &stb);// Reads the Status Byte
sweep=0;// Sets the flag to stop the '*' printing
printf("\n");// Print user information
printf("An SRQ, indicating end of sweep has occurred\n");
viClose(event);// Closes the event
return VI_SUCCESS;
}
```

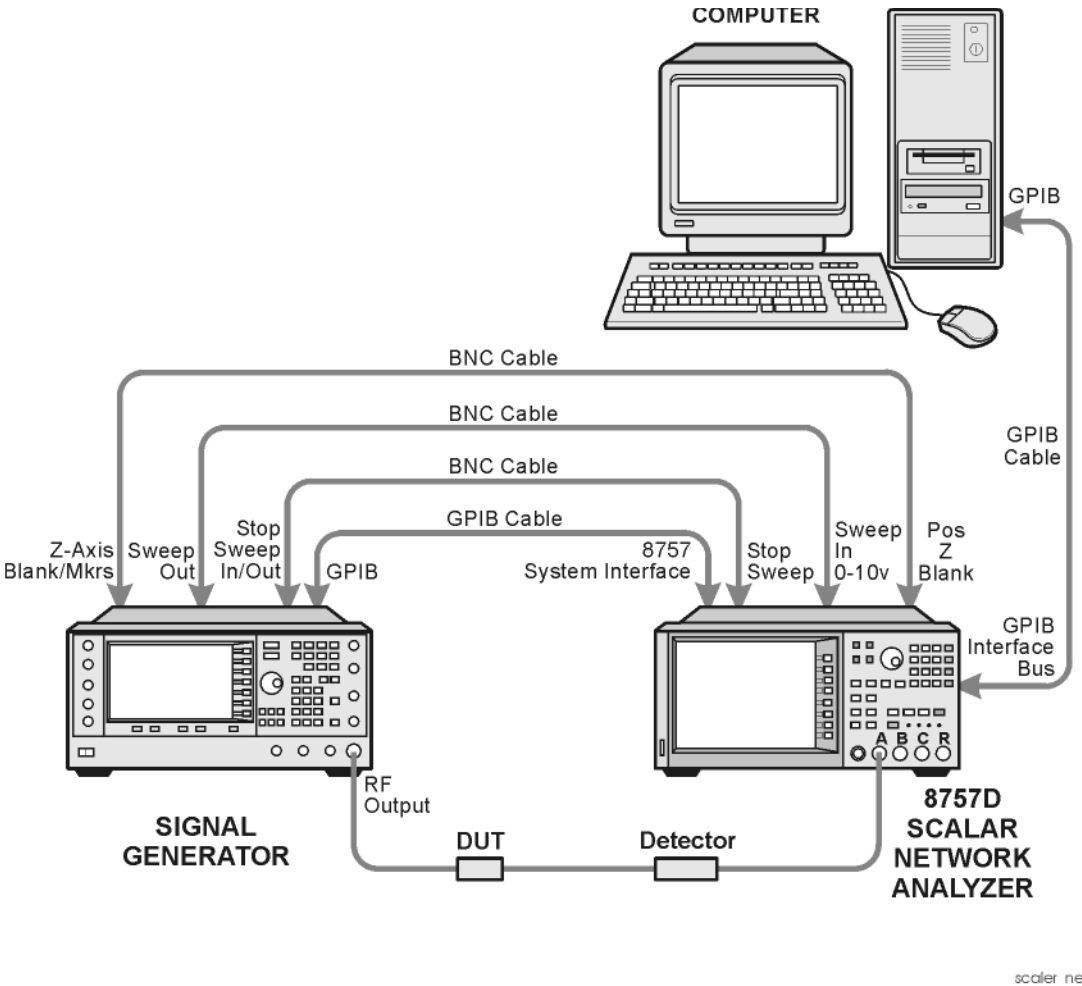
Using 8757D Pass-Thru Commands (PSG with Option 007 Only)

Pass-thru commands enable you to send operating instructions to a PSG that is connected to a 8757D scalar analyzer system. This section provides setup information and an example program for using pass-thru commands in a ramp sweep system.

Equipment Setup

To send pass-thru commands, set up the equipment as shown in Figure 3-1. Notice that the GPIB cable from the computer is connected to the GPIB interface bus of the 8757D. The GPIB cable from the PSG is connected to the system interface bus of the 8757D.

Figure 3-1



GPIB Address Assignments

Figure 3-1 describes how GPIB addresses should be assigned for sending pass-thru commands. These are the same addresses used in Example 3-1.

Table 3-1

Instrument	GPIB Address	Key Presses/Description
PSG/E8663B	19	Press **Utility** > **GPIB/RS-232 LAN** > **GPIB Address** > **19** > **Enter**.
8757D	16	Press **LOCAL** > **8757** > **16** > **Enter**.
8757D (Sweeper)	19	This address must match the PSG. Press **LOCAL** > **SWEEPER** > **19** > **Enter**.
Pass Thru	17	The pass thru address is automatically selected by the 8757D by inverting the last bit of the 8757D address. Refer to the 8757D documentation for more information. Verify that no other instrument is using this address on the GPIB bus.

Example Pass-Thru Program

Example 3-1 on page 110 is a sample Agilent BASIC program that switches the 8757D to pass-thru mode, allowing you to send operating commands to the PSG. After the program runs, control is given back to the network analyzer. The following describes the command lines used in the program.

Line 30 — PT is set to equal the source address. C1 is added, but not needed, to specify the channel.

Lines 40, 90 — The END statement is required to complete the language transition.

Lines 50, 100 — A WAIT statement is recommended after a language change to allow all instrument changes to be completed before the next command.

Lines 70, 80 — This is added to ensure that the instrument has completed all operations before switching languages. Lines 70 and 80 can only be used when the signal generator is in single sweep mode.

Line 110 — This takes the network analyzer out of pass-thru command mode, and puts it back in control. Any analyzer command can now be entered.

NOTE Verify the signal generator is in single sweep mode. Refer to the *SCPI Reference* or the *User's Guide*, as required.

Example 3-1 Pass-Thru Program

```
10 ABORT 7
20 CLEAR 716
30 OUTPUT 716;"PT19;C1"
```

```
40 OUTPUT 717;"SYST:LANG SCPI";END
50 WAIT .5
60 OUTPUT 717;"OUTP:STAT OFF"
70 OUTPUT 717;"*OPC?"
80 ENTER 717; Reply
90 OUTPUT 717;"SYST:LANG COMP";END
100 WAIT .5
110 OUTPUT 716;"C2"
120 END
```

Setting the PSG Sweep Time Requirements (PSG with Firmware ≥4.92)

By default, the PSG sweep time is automatically adjusted to the fastest possible sweep when exiting Pass-Thru mode. To disable this feature and retain specific time selection, set the 8757D system time to Manual.

The front panel key sequence is:

Sweep > **Sweep Type** > **8757D System Sweep Time** to Manual

The SCPI command sequence is:

```
[:SOURce]:SWEep:TIME:COMP:AUTO OFF
```

NOTE The changes to sweep time mode are persistent. For more information on persistent memory, refer to the *User's Guide*.

The SCPI command to set up sweep time is: `:SWE:TIME <val><units>`.

For more on SCPI commands, refer to the *SCPI Command Reference*.

Setting the PSG Sweep Time Requirements (PSG with Firmware ≤4.91)

By default, the PSG sweep time is automatically adjusted to the fastest possible sweep when exiting Pass-Thru mode. To select a different sweep time, these additional steps are required:

1. Insert line 25, that saves state 1 (SV1).

   ```
   25 OUTPUT 716;"SV1"
   ```

2. Insert line 55, that sets the sweep-time of the source, :SWE:TIME <val>.

   ```
   55 OUTPUT 717;":SWE:TIME .200S"
   ```

3. Insert line 56, that saves the state into the register, sequence 0, register 1, *SAV <reg_num>[,<seq_num>], (`*SAV 1,0`).

   ```
   56 OUTPUT 717;"*SAV 1,0"
   ```

4. Insert line 115, that recalls state 1, (RC1).

```
115 OUTPUT 717;"RC1"
```

LAN Programming Interface Examples

NOTE The LAN programming examples in this section demonstrate the use of VXI-11 and Sockets LAN to control the signal generator.

To use these programming examples you must change references to the IP address and hostname to match the IP address and hostname of your signal generator.

- "VXI-11 Programming Using SICL and C++" on page 113
- "VXI-11 Programming Using VISA and C++" on page 114
- "Sockets LAN Programming and C" on page 116
- "Sockets LAN Programming Using Java" on page 139
- "Sockets LAN Programming Using PERL" on page 141

For additional LAN programming examples that work with user-data files, refer to:

- "Save and Recall Instrument State Files" on page 336

VXI-11 Programming

The signal generator supports the VXI-11 standard for instrument communication over the LAN interface. Agilent IO Libraries support the VXI-11 standard and must be installed on your computer before using the VXI-11 protocol. Refer to "Using VXI–11" on page 39 for information on configuring and using the VXI-11 protocol.

The VXI-11 examples use TCPIP0 as the board address.

Using VXI-11 with GPIB Programs

The GPIB programming examples that use the VISA library, and are listed in "GPIB Programming Interface Examples" on page 74, can be easily changed to use the LAN VXI-11 protocol by changing the address string. For example, change the "GPIB::19::INSTR" address string to "TCPIP::hostname::INSTR" where hostname is the IP address or hostname of the signal generator. The VXI-11 protocol has the same capabilities as GPIB. See the section "Setting Up the LAN Interface" on page 30 for more information.

NOTE To communicate with the signal generator over the LAN interface you must enable the VXI-11 SCPI service. For more information, refer to "Configuring the VXI–11 for LAN (Agilent MXG)" on page 30 and "Configuring the VXI–11 for LAN (ESG/PSG/E8663B)" on page 31.

VXI-11 Programming Using SICL and C++

The following program uses the VXI-11 protocol and SICL to control the signal generator. Before running this code, you must set up the interface using the Agilent IO Libraries IO Config utility. `vxisicl.cpp` performs the following functions:

- sets signal generator to 1 GHz CW frequency
- queries signal generator for an ID string
- error checking

The following program example is available on the signal generator Documentation CD-ROM as `vxisicl.cpp`.

```
//**************************************************************************************
//
// PROGRAM NAME:vxisicl.cpp
//
// PROGRAM DESCRIPTION:Sample test program using SICL and the VXI-11 protocol
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the VXI-11 protocol to set the signal generator for a 1 gHz CW // frequency. The
signal generator is queried for operation complete and then queried
// for its ID string. The frequency and ID string are then printed to the display.
//
// IMPORTANT: Enter in your signal generators hostname in the instrumentName declaration
// where the "xxxxx" appears.
//
//**************************************************************************************

#include "stdafx.h"
#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[])
{

INST id;                                    // Device session id
int opcResponse;                            // Variable for response flag

char instrumentName[] = "xxxxx"; // Put your instrument's hostname here
char instNameBuf[256];// Variable to hold instrument name
char buf[256];// Variable for id string
ionerror(I_ERROR_EXIT);// Register SICL error handler
```

```
                // Open SICL instrument handle using VXI-11 protocol

sprintf(instNameBuf, "lan[%s]:inst0", instrumentName);
id = iopen(instNameBuf);// Open instrument session
itimeout(id, 1000);// Set 1 second timeout for operations
printf("Setting frequency to 1 Ghz...\n");
iprintf(id, "freq 1 GHz\n");// Set frequency to 1 GHz

printf("Waiting for source to settle...\n");
iprintf(id, "*opc?\n");// Query for operation complete
iscanf(id, "%d", &opcResponse);  // Operation complete flag
if (opcResponse != 1)// If operation fails, prompt user
 {
   printf("Bad response to 'OPC?'\n");
   iclose(id);
   exit(1);
}
iprintf(id, "FREQ?\n");// Query the frequency
iscanf(id, "%t", &buf);// Read the signal generator frequency
printf("\n");// Print the frequency to the display
printf("Frequency of signal generator is  %s\n", buf);
ipromptf(id, "*IDN?\n", "%t", buf);// Query for id string
printf("Instrument ID: %s\n", buf);// Print id string to display
iclose(id);// Close the session

return 0;
}
```

VXI-11 Programming Using VISA and C++

The following program uses the VXI-11 protocol and the VISA library to control the signal generator. The signal generator is set to a –5 dBm power level and queried for its ID string. Before running this code, you must set up the interface using the Agilent IO Libraries IO Config utility. `vxivisa.cpp` performs the following functions:

- sets signal generator to a –5 dBm power level
- queries signal generator for an ID string
- error checking

The following program example is available on the signal generator Documentation CD-ROM as `vxivisa.cpp`.

```
//*****************************************************************************
// PROGRAM FILE NAME:vxivisa.cpp
// Sample test program using the VISA libraries and the VXI-11 protocol
//
```

```
// NOTE: You must have the Agilent Libraries installed on your computer to run
// this program
//
// PROGRAM DESCRIPTION:This example uses the VXI-11 protocol and VISA to query
// the signal generator for its ID string. The ID string is then printed to the
// screen. Next the signal generator is set for a -5 dBm power level and then
// queried for the power level. The power level is printed to the screen.
//
// IMPORTANT: Set up the LAN Client using the IO Config utility
//
//*******************************************************************************

#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>
#include <conio.h>

#define MAX_COUNT 200

int main (void)

{

ViStatus status;// Declares a type ViStatus variable
ViSession defaultRM, instr;// Declares a type ViSession variable
ViUInt32 retCount;// Return count for string I/O
ViChar buffer[MAX_COUNT];// Buffer for string I/O

status = viOpenDefaultRM(&defaultRM);    // Initialize the system
                                         // Open communication with Serial
                                         // Port 2
status = viOpen(defaultRM, "TPCIP0::19::INSTR", VI_NULL, VI_NULL, &instr);

if(status){                              // If problems then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}
                                             // Set timeout for 5 seconds
viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);
                                         // Ask for sig gen ID string
```

```
 status = viWrite(instr, (ViBuf)"*IDN?\n", 6, &retCount);

                                        // Read the sig gen response
 status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';                 // Indicate the end of the string
printf("Signal Generator ID = ");       // Print header for ID
printf(buffer);                         // Print the ID string
printf("\n");                           // Print carriage return
                                        // Flush the read buffer
                                        // Set sig gen power to -5dbm
status = viWrite(instr, (ViBuf)"POW:AMPL -5dbm\n", 15, &retCount);
                                        // Query the power level
status = viWrite(instr, (ViBuf)"POW?\n",5,&retCount);
                                        // Read the power level
status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';                 // Indicate the end of the string
printf("Power level = ");               // Print header to the screen
printf(buffer);                         // Print the queried power level
printf("\n");
status = viClose(instr);                // Close down the system
status = viClose(defaultRM);
return 0;
}
```

Sockets LAN Programming and C

The program listing shown in "Queries for Lan Using Sockets" on page 119 consists of two files; lanio.c and getopt.c. The lanio.c file has two main functions; `int main()` and an `int main1()`.

The `int main()` function allows communication with the signal generator interactively from the command line. The program reads the signal generator's hostname from the command line, followed by the SCPI command. It then opens a socket to the signal generator, using port 5025, and sends the command. If the command appears to be a query, the program queries the signal generator for a response, and prints the response.

The `int main1()`, after renaming to `int main()`, will output a sequence of commands to the signal generator. You can use the format as a template and then add your own code.

This program is available on the signal generator Documentation CD-ROM as `lanio.c`.

Sockets on UNIX

In UNIX, LAN communication via sockets is very similar to reading or writing a file. The only difference is the `openSocket()` routine, which uses a few network library routines to create the TCP/IP network connection. Once this connection is created, the `standard fread()` and `fwrite()` routines are used for network communication. The following steps outline the process:

1. Copy the lanio.c and getopt.c files to your home UNIX directory. For example, `/users/mydir/`.

2. At the UNIX prompt in your home directory type: `cc -Aa -O -o lanio lanio.c`
3. At the UNIX prompt in your home directory type: `./lanio xxxxx "*IDN?"` where `xxxxx` is the hostname for the signal generator. Use this same format to output SCPI commands to the signal generator.

The `int main1()` function will output a sequence of commands in a program format. If you want to run a program using a sequence of commands then perform the following:

1. Rename the lanio.c `int main1()` to `int main()` and the original `int main()` to `int main1()`.
2. In the `main()`, `openSocket()` function, change the "your hostname here" string to the hostname of the signal generator you want to control.
3. Re-save the lanio.c program.
4. At the UNIX prompt type: `cc -Aa -O -o lanio lanio.c`
5. At the UNIX prompt type: `./lanio`

The program will run and output a sequence of SCPI commands to the signal generator. The UNIX display will show a display similar to the following:

```
unix machine: /users/mydir
$ ./lanio
ID: Agilent Technologies, E4438C, US70000001, C.02.00

Frequency: +2.5000000000000E+09
Power Level: -5.00000000E+000
```

Sockets on Windows

In Windows, the routines send() and recv() must be used, since fread() and fwrite() may not work on sockets. The following steps outline the process for running the interactive program in the Microsoft Visual C++ 6.0 environment:

1. Rename the lanio.c to lanio.cpp and getopt.c to getopt.cpp and add them to the Source folder of the Visual C++ project.

NOTE The int main() function in the lanio.cpp file will allow commands to be sent to the signal generator in a line-by-line format; the user types in SCPI commands. The int main1(0) function can be used to output a sequence of commands in a "program format." See Programming Using main1() Function below.

---

2. Click **Rebuild All** from **Build** menu. Then Click **Execute Lanio.exe**. The Debug window will appear with a prompt "Press any key to continue." This indicates that the program has compiled and can be used to send commands to the signal generator.
3. Click **Start**, click **Programs**, then click **Command Prompt**. The command prompt window will appear.
4. At the command prompt, `cd` to the directory containing the lanio.exe file and then to the Debug folder. For example C:\SocketIO\Lanio\Debug.

5. After you cd to the directory where the lanio.exe file is located, type in the following command at the command prompt: `lanio xxxxx "*IDN?"`. For example: `C:\SocketIO\Lanio\Debug>lanio xxxxx "*IDN?"` where the xxxxx is the hostname of your signal generator. Use this format to output SCPI commands to the signal generator in a line by line format from the command prompt.
6. Type `exit` at the command prompt to quit the program.

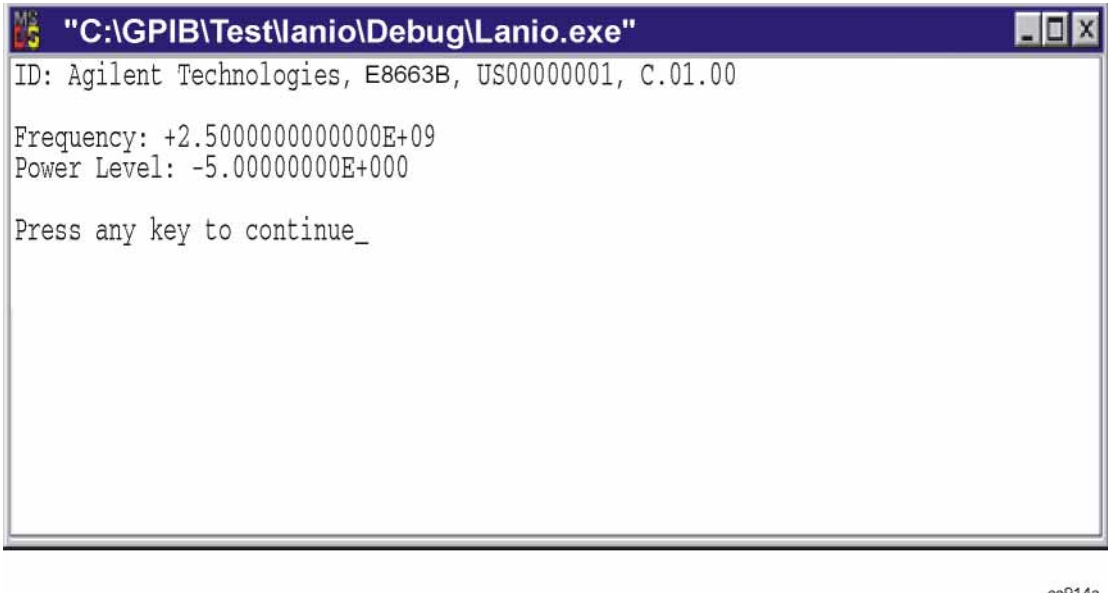
Programming Using main1() Function

The `int main1()` function will output a sequence of commands in a program format. If you want to run a program using a sequence of commands then perform the following:

1. Enter the hostname of your signal generator in the openSocket function of the `main1()` function of the lanio.cpp program.
2. Rename the lanio.cpp `int main1()` function to `int main()` and the original `int main()` function to `int main1()`.
3. Select **Rebuild All** from **Build** menu. Then select **Execute Lanio.exe**.

The program will run and display results similar to those shown in Figure 3-2.

Figure 3-2 Program Output Screen



Queries for Lan Using Sockets

lanio.c and getopt.c perform the following functions:

- establishes TCP/IP connection to port 5025
- resultant file descriptor is used to “talk” to the instrument using regular socket I/O mechanisms
- maps the desired hostname to an internal form
- error checks
- queries signal generator for ID
- sets frequency on signal generator to 2.5 GHz
- sets power on signal generator to –5 dBm
- gets option letter from argument vector and checks for end of file (EOF)

The following programming examples are available on the signal generator Documentation CD-ROM as lanio.c and getopt.c.

```
/*************************************************************************
*  $Header: lanio.c 04/24/01
*  $Revision: 1.1 $
*  $Date: 10/24/01
*  PROGRAM NAME:   lanio.c
*
*  $Description:     Functions to talk to an Agilent signal generator
*                    via TCP/IP.  Uses command-line arguments.
*
*                    A TCP/IP connection to port 5025 is established and
*                    the resultant file descriptor is used to "talk" to the
*                    instrument using regular socket I/O mechanisms. $
*
*
*
*  Examples:
*
*   Query the signal generator frequency:
*        lanio xx.xxx.xx.x 'FREQ?'
*
*   Query the signal generator power level:
*        lanio xx.xxx.xx.x  'POW?'
*
*   Check for errors (gets one error):
*        lanio xx.xxx.xx.x  'syst:err?'
*
*   Send a list of commands from a file, and number them:
*        cat scpi_cmds | lanio -n xx.xxx.xx.x
*
*************************************************************************
```

```
 *
 *  This program compiles and runs under
 *     - HP-UX 10.20 (UNIX), using HP cc or gcc:
 *          + cc -Aa    -O -o lanio  lanio.c
 *          + gcc -Wall -O -o lanio  lanio.c
 *
 *     - Windows 95, using Microsoft Visual C++ 4.0 Standard Edition
 *     - Windows NT 3.51, using Microsoft Visual C++ 4.0
 *          + Be sure to add  WSOCK32.LIB  to your list of libraries!
 *          + Compile both lanio.c and getopt.c
 *          + Consider re-naming the files to lanio.cpp and getopt.cpp
 *
 *  Considerations:
 *     - On UNIX systems, file I/O can be used on network sockets.
 *       This makes programming very convenient, since routines like
 *       getc(), fgets(), fscanf() and fprintf() can be used.  These
 *       routines typically use the lower level read() and write() calls.
 *
 *     - In the Windows environment, file operations such as read(), write(),
 *       and close() cannot be assumed to work correctly when applied to
 *       sockets.  Instead, the functions send() and recv() MUST be used.
 ******************************************************************************/

/* Support both Win32 and HP-UX UNIX environment */

#ifdef _WIN32     /* Visual C++ 6.0 will define this */
#  define WINSOCK
#endif

#ifndef WINSOCK
#  ifndef _HPUX_SOURCE
#  define _HPUX_SOURCE
#  endif
#endif

#include <stdio.h>         /* for fprintf and NULL  */
#include <string.h>        /* for memcpy and memset */
#include <stdlib.h>        /* for malloc(), atol() */
#include <errno.h>         /* for strerror          */

#ifdef WINSOCK
```

```
#include <windows.h>

#  ifndef _WINSOCKAPI_
#  include <winsock.h>   // BSD-style socket functions
#  endif

#else                         /* UNIX with BSD sockets */

#  include <sys/socket.h>    /* for connect and socket*/
#  include <netinet/in.h>    /* for sockaddr_in       */
#  include <netdb.h>         /* for gethostbyname     */

#  define SOCKET_ERROR (-1)
#  define INVALID_SOCKET (-1)

   typedef  int SOCKET;

#endif /* WINSOCK */

#ifdef WINSOCK
  /* Declared in getopt.c.  See example programs disk. */
  extern char *optarg;
  extern int  optind;
  extern int getopt(int argc, char * const argv[], const char* optstring);
#else
#  include <unistd.h>           /* for getopt(3C) */
#endif

#define COMMAND_ERROR  (1)
#define NO_CMD_ERROR  (0)

#define SCPI_PORT  5025
#define INPUT_BUF_SIZE (64*1024)



/*************************************************************************
 * Display usage
 ************************************************************************/
static void usage(char *basename)
{
    fprintf(stderr,"Usage: %s [-nqu] <hostname> [<command>]\n", basename);
```

```
    fprintf(stderr,"       %s [-nqu] <hostname> < stdin\n", basename);
    fprintf(stderr,"  -n, number output lines\n");
    fprintf(stderr,"  -q, quiet; do NOT echo lines\n");
    fprintf(stderr,"  -e, show messages in error queue when done\n");
}



#ifdef WINSOCK
int init_winsock(void)
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD(1, 1);
    wVersionRequested = MAKEWORD(2, 0);

    err = WSAStartup(wVersionRequested, &wsaData);

    if (err != 0) {
        /* Tell the user that we couldn't find a useable */
        /* winsock.dll.     */
        fprintf(stderr, "Cannot initialize Winsock 1.1.\n");
        return -1;
    }
    return 0;
}

int close_winsock(void)
{
    WSACleanup();
    return 0;
}
#endif /* WINSOCK */



/**************************************************************************
 *
 > $Function: openSocket$
 *
 * $Description:  open a TCP/IP socket connection to the instrument $
```

```
 *
 * $Parameters:  $
 *    (const char *) hostname . . . . Network name of instrument.
 *                                    This can be in dotted decimal notation.
 *    (int) portNumber  . . . . . . . The TCP/IP port to talk to.
 *                                    Use 5025 for the SCPI port.
 *
 * $Return:     (int)  . . . . . . . . A file descriptor similar to open(1).$
 *
 * $Errors:     returns -1 if anything goes wrong $
 *
 ******************************************************************************/
SOCKET openSocket(const char *hostname, int portNumber)
{
    struct hostent *hostPtr;
    struct sockaddr_in peeraddr_in;
    SOCKET s;


    memset(&peeraddr_in, 0, sizeof(struct sockaddr_in));

    /*********************************************/
    /* map the desired host name to internal form. */
    /*********************************************/
    hostPtr = gethostbyname(hostname);
    if (hostPtr == NULL)
    {
        fprintf(stderr,"unable to resolve hostname '%s'\n", hostname);
        return INVALID_SOCKET;
    }

    /******************/
    /* create a socket */
    /******************/
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == INVALID_SOCKET)
    {
        fprintf(stderr,"unable to create socket to '%s': %s\n",
                hostname, strerror(errno));
        return INVALID_SOCKET;
    }
```

```
    memcpy(&peeraddr_in.sin_addr.s_addr, hostPtr->h_addr, hostPtr->h_length);
    peeraddr_in.sin_family = AF_INET;
    peeraddr_in.sin_port = htons((unsigned short)portNumber);

    if (connect(s, (const struct sockaddr*)&peeraddr_in,
                sizeof(struct sockaddr_in)) == SOCKET_ERROR)
    {
        fprintf(stderr,"unable to create socket to '%s': %s\n",
                hostname, strerror(errno));
        return INVALID_SOCKET;
    }

    return s;
}



/*************************************************************************
 *
 > $Function: commandInstrument$
 *
 * $Description:  send a SCPI command to the instrument.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 * $Return:  (char *) . . . . . a pointer to the result string.
 *
 * $Errors:   returns 0 if send fails $
 *
 *************************************************************************/
int commandInstrument(SOCKET sock,
                      const char *command)
{
    int count;

    /* fprintf(stderr, "Sending \"%s\".\n", command);  */
    if (strchr(command, '\n') == NULL) {
       fprintf(stderr, "Warning: missing newline on command %s.\n", command);
    }

    count = send(sock, command, strlen(command), 0);
```

```
    if (count == SOCKET_ERROR) {
        return COMMAND_ERROR;
    }

    return NO_CMD_ERROR;
}


/*************************************************************************
 * recv_line(): similar to fgets(), but uses recv()
 *************************************************************************/
char * recv_line(SOCKET sock, char * result, int maxLength)
{
#ifdef WINSOCK
    int cur_length = 0;
    int count;
    char * ptr = result;
    int err = 1;

    while (cur_length < maxLength) {
        /* Get a byte into ptr */
        count = recv(sock, ptr, 1, 0);

        /* If no chars to read, stop. */
        if (count < 1) {
            break;
        }
        cur_length += count;

        /* If we hit a newline, stop. */
        if (*ptr == '\n') {
            ptr++;
            err = 0;
            break;
        }
        ptr++;

    }

    *ptr = '\0';

    if (err) {
```

```
        return NULL;
    } else {
        return result;
    }
#else
    /*************************************************************************
     * Simpler UNIX version, using file I/O.  recv() version works too.
     * This demonstrates how to use file I/O on sockets, in UNIX.
     ************************************************************************/
    FILE * instFile;
    instFile = fdopen(sock, "r+");
    if (instFile == NULL)
    {
        fprintf(stderr, "Unable to create FILE * structure : %s\n",
                strerror(errno));
        exit(2);
    }
    return fgets(result, maxLength, instFile);
#endif
}



/*************************************************************************
 *
 > $Function: queryInstrument$
 *
 * $Description:  send a SCPI command to the instrument, return a response.$
 *
 * $Parameters:  $
 *     (FILE *) . . . . . file pointer associated with TCP/IP socket.
 *     (const char *command)  . . SCPI command string.
 *     (char *result) . . . . . where to put the result.
 *     (size_t) maxLength . . . . maximum size of result array in bytes.
 *
 * $Return:  (long) . . . . . The number of bytes in result buffer.
 *
 * $Errors:   returns 0 if anything goes wrong. $
 *
 ************************************************************************/
long queryInstrument(SOCKET sock,
                     const char *command, char *result, size_t maxLength)
```

```
{
    long ch;
    char tmp_buf[8];
    long resultBytes = 0;
    int command_err;
    int count;

    /*********************************************************
     * Send command to signal generator
     *********************************************************/
    command_err = commandInstrument(sock, command);
    if (command_err) return COMMAND_ERROR;


    /*********************************************************
     * Read response from signal generator
     *********************************************************/
    count = recv(sock, tmp_buf, 1, 0); /* read 1 char */
    ch = tmp_buf[0];

    if ((count < 1) || (ch == EOF)  || (ch == '\n'))
    {
        *result = '\0';  /* null terminate result for ascii */
        return 0;
    }

    /* use a do-while so we can break out */
    do
    {
        if (ch == '#')
        {
            /* binary data encountered - figure out what it is */
            long numDigits;
            long numBytes = 0;
            /* char length[10]; */

            count = recv(sock, tmp_buf, 1, 0); /* read 1 char */
            ch = tmp_buf[0];
            if ((count < 1) || (ch == EOF)) break; /* End of file */

            if (ch < '0' || ch > '9') break;  /* unexpected char */
            numDigits = ch - '0';
```

```
if (numDigits)
{
    /* read numDigits bytes into result string. */
    count = recv(sock, result, (int)numDigits, 0);
    result[count] = 0;  /* null terminate */
    numBytes = atol(result);
}

if (numBytes)
{
    resultBytes = 0;
    /* Loop until we get all the bytes we requested. */
    /* Each call seems to return up to 1457 bytes, on HP-UX 9.05 */
    do {
        int rcount;
        rcount = recv(sock, result, (int)numBytes, 0);
        resultBytes += rcount;
        result      += rcount;  /* Advance pointer */
    } while ( resultBytes < numBytes );

    /*************************************************************
     * For LAN dumps, there is always an extra trailing newline
     * Since there is no EOI line.  For ASCII dumps this is
     * great but for binary dumps, it is not needed.
     *************************************************************/
    if (resultBytes == numBytes)
    {
        char junk;
        count = recv(sock, &junk, 1, 0);
    }
}
else
{
    /* indefinite block ... dump til we can an extra line feed */
    do
    {
        if (recv_line(sock, result, maxLength) == NULL) break;
        if (strlen(result)==1 && *result == '\n') break;
        resultBytes += strlen(result);
        result += strlen(result);
    } while (1);
```

```
            }
        }
        else
        {
            /* ASCII response (not a binary block) */
            *result = (char)ch;
            if (recv_line(sock, result+1, maxLength-1) == NULL) return 0;

            /* REMOVE trailing newline, if present.  And terminate string. */
            resultBytes = strlen(result);
            if (result[resultBytes-1] == '\n') resultBytes -= 1;
            result[resultBytes] = '\0';
        }
    } while (0);

    return resultBytes;
}




/*************************************************************************
 *
 > $Function: showErrors$
 *
 * $Description: Query the SCPI error queue, until empty.  Print results. $
 *
 * $Return:  (void)
 *
 *************************************************************************/
void showErrors(SOCKET sock)
{
    const char * command = "SYST:ERR?\n";
    char result_str[256];

    do {
        queryInstrument(sock, command, result_str, sizeof(result_str)-1);

        /*****************************************************************
         * Typical result_str:
         *     -221,"Settings conflict; Frequency span reduced."
         *     +0,"No error"
```

```
         * Don't bother decoding.
         ********************************************************************/
        if (strncmp(result_str, "+0,", 3) == 0) {
            /* Matched +0,"No error" */
            break;
        }
        puts(result_str);
    } while (1);

}


/*****************************************************************************
 *
 > $Function: isQuery$
 *
 * $Description: Test current SCPI command to see if it a query. $
 *
 * $Return:  (unsigned char) . . . non-zero if command is a query.  0 if not.
 *
 *****************************************************************************/
unsigned char isQuery( char* cmd )
{
    unsigned char q = 0 ;
    char *query ;

    /*********************************************************/
    /* if the command has a '?' in it, use queryInstrument.  */
    /* otherwise, simply send the command.                   */
    /* Actually, we must be a more specific so that   */
    /* marker value querys are treated as commands.         */
    /* Example:  SENS:FREQ:CENT (CALC1:MARK1:X?)             */
    /*********************************************************/
    if ( (query = strchr(cmd,'?')) != NULL)
    {
        /* Make sure we don't have a marker value query, or
         * any command with a '?' followed by a ')' character.
         * This kind of command is not a query from our point of view.
         * The signal generator does the query internally, and uses the result.
         */
        query++ ;       /* bump past '?' */
        while (*query)
```

```
        {
            if (*query == ' ') /* attempt to ignore white spc */
                query++ ;
            else break ;
        }

        if ( *query != ')' )
        {
            q = 1 ;
        }
    }
    return q ;
}

/*****************************************************************************
 *
 > $Function: main$
 *
 * $Description: Read command line arguments, and talk to signal generator.
                 Send query results to stdout. $
 *
 * $Return:  (int) . . . non-zero if an error occurs
 *
 *****************************************************************************/

int main(int argc, char *argv[])
{

    SOCKET instSock;
    char *charBuf = (char *) malloc(INPUT_BUF_SIZE);
    char *basename;
    int chr;
    char command[1024];
    char *destination;
    unsigned char quiet = 0;
    unsigned char show_errs = 0;
    int number = 0;

    basename = strrchr(argv[0], '/');
    if (basename != NULL)
        basename++ ;
    else
```

```
    basename = argv[0];

while ( ( chr = getopt(argc,argv,"qune")) != EOF )
    switch (chr)
    {
        case 'q':  quiet = 1; break;
        case 'n':  number = 1; break ;
        case 'e':  show_errs = 1; break ;
        case 'u':
        case '?':  usage(basename); exit(1) ;
    }

/* now look for hostname and optional <command>*/
if (optind < argc)
{
    destination = argv[optind++] ;
    strcpy(command, "");
    if (optind < argc)
    {
        while (optind < argc) {
            /* <hostname> <command> provided; only one command string */
            strcat(command, argv[optind++]);
            if (optind < argc) {
                strcat(command, " ");
            } else {
                strcat(command, "\n");
            }
        }
    }
    else
    {
        /*Only <hostname> provided; input on <stdin> */
        strcpy(command, "");

        if (optind > argc)
        {
            usage(basename);
            exit(1);
        }
    }
}
else
```

```
    {
        /* no hostname! */
        usage(basename);
        exit(1);
    }

    /*****************************************************
    /* open a socket connection to the instrument
    /*****************************************************/

#ifdef WINSOCK
    if (init_winsock() != 0) {
        exit(1);
    }
#endif /* WINSOCK */

    instSock = openSocket(destination, SCPI_PORT);
    if (instSock == INVALID_SOCKET) {
        fprintf(stderr, "Unable to open socket.\n");
        return 1;
    }
    /* fprintf(stderr, "Socket opened.\n"); */

    if (strlen(command) > 0)
    {
    /*****************************************************
    /* if the command has a '?' in it, use queryInstrument. */
    /* otherwise, simply send the command.                  */
    /*****************************************************/
        if ( isQuery(command) )
        {
            long bufBytes;
            bufBytes = queryInstrument(instSock, command,
                                       charBuf, INPUT_BUF_SIZE);
            if (!quiet)
            {
                fwrite(charBuf, bufBytes, 1, stdout);
                fwrite("\n", 1, 1, stdout) ;
                fflush(stdout);
            }
        }
        else
```

```
        {
            commandInstrument(instSock, command);
        }
    }
    else
    {
        /* read a line from <stdin> */
        while ( gets(charBuf) != NULL )
        {
            if ( !strlen(charBuf) )
                continue ;

            if ( *charBuf == '#' || *charBuf == '!' )
                continue ;

            strcat(charBuf, "\n");

            if (!quiet)
            {
                if (number)
                {
                    char num[10];
                    sprintf(num,"%d: ",number);
                    fwrite(num, strlen(num), 1, stdout);
                }
                fwrite(charBuf, strlen(charBuf), 1, stdout) ;
                fflush(stdout);
            }

            if ( isQuery(charBuf) )
            {
                long bufBytes;

                /* Put the query response into the same buffer as the*/
                /* command string appended after the null terminator.*/

                bufBytes = queryInstrument(instSock, charBuf,
                                           charBuf + strlen(charBuf) + 1,
                                           INPUT_BUF_SIZE -strlen(charBuf) );
                if (!quiet)
                {
                    fwrite("  ", 2, 1, stdout) ;
```

```
                    fwrite(charBuf + strlen(charBuf)+1, bufBytes, 1, stdout);
                    fwrite("\n", 1, 1, stdout) ;
                    fflush(stdout);
                }
            }
            else
            {
                commandInstrument(instSock, charBuf);
            }
            if (number) number++;
        }
    }

    if (show_errs) {
        showErrors(instSock);
    }

#ifdef WINSOCK
    closesocket(instSock);
    close_winsock();
#else
    close(instSock);
#endif /* WINSOCK */

    return 0;
}

/* End of lanio.cpp  *



/*************************************************************************/
/* $Function: main1$                                                     */
/* $Description: Output a series of SCPI commands to the signal generator */
/*               Send query results to stdout. $                         */
/*                                                                       */
/* $Return:  (int) . . . non-zero if an error occurs                     */
/*                                                                       */
/*************************************************************************/
/* Rename this int main1() function to int main(). Re-compile and the    */
/* execute the program                                                   */
/*************************************************************************/
```

```
int main1()
{

SOCKET instSock;
long bufBytes;
    char *charBuf = (char *) malloc(INPUT_BUF_SIZE);


    /*********************************************/
    /* open a socket connection to the instrument*/
    /*********************************************/

#ifdef WINSOCK
    if (init_winsock() != 0) {
        exit(1);
    }
#endif /* WINSOCK */

    instSock = openSocket("xxxxxx", SCPI_PORT); /* Put your hostname here */
    if (instSock == INVALID_SOCKET) {
        fprintf(stderr, "Unable to open socket.\n");
        return 1;
    }
    /* fprintf(stderr, "Socket opened.\n"); */

   bufBytes = queryInstrument(instSock, "*IDN?\n", charBuf, INPUT_BUF_SIZE);
   printf("ID: %s\n",charBuf);
   commandInstrument(instSock, "FREQ 2.5 GHz\n");
   printf("\n");
   bufBytes = queryInstrument(instSock, "FREQ:CW?\n", charBuf, INPUT_BUF_SIZE);
   printf("Frequency: %s\n",charBuf);
   commandInstrument(instSock, "POW:AMPL -5 dBm\n");
   bufBytes = queryInstrument(instSock, "POW:AMPL?\n", charBuf, INPUT_BUF_SIZE);
   printf("Power Level: %s\n",charBuf);
   printf("\n");


#ifdef WINSOCK
    closesocket(instSock);
    close_winsock();
#else
```

```
    close(instSock);
#endif /* WINSOCK */

    return 0;
}
/*****************************************************************************

 getopt(3C)                                                        getopt(3C)

 PROGRAM FILE NAME: getopt.c
 getopt - get option letter from argument vector

 SYNOPSIS
      int getopt(int argc, char * const argv[], const char *optstring);
      extern char *optarg;
      extern int optind, opterr, optopt;

 PRORGAM DESCRIPTION:
      getopt returns the next option letter in argv (starting from argv[1])
      that matches a letter in optstring.  optstring is a string of
      recognized option letters; if a letter is followed by a colon, the
      option is expected to have an argument that may or may not be
      separated from it by white space.  optarg is set to point to the start
      of the option argument on return from getopt.

      getopt places in optind the argv index of the next argument to be
      processed.  The external variable optind is initialized to 1 before
      the first call to the function getopt.

      When all options have been processed (i.e., up to the first non-option
      argument), getopt returns EOF.  The special option -- can be used to
      delimit the end of the options; EOF is returned, and -- is skipped.

 *****************************************************************************/


#include <stdio.h>      /* For NULL, EOF */
#include <string.h>     /* For strchr() */

char    *optarg;        /* Global argument pointer. */
int     optind = 0;     /* Global argv index. */
```

```
static char     *scan = NULL;   /* Private scan pointer. */

int getopt( int argc, char * const argv[], const char* optstring)
{
    char c;
    char *posn;

    optarg = NULL;

    if (scan == NULL || *scan == '\0') {
        if (optind == 0)
            optind++;

        if (optind >= argc || argv[optind][0] != '-' || argv[optind][1] == '\0')
            return(EOF);
        if (strcmp(argv[optind], "--")==0) {
            optind++;
            return(EOF);
        }

        scan = argv[optind]+1;
        optind++;
    }

    c = *scan++;
    posn = strchr(optstring, c);        /* DDP */

    if (posn == NULL || c == ':') {
        fprintf(stderr, "%s: unknown option -%c\n", argv[0], c);
        return('?');
    }

    posn++;
    if (*posn == ':') {
        if (*scan != '\0') {
            optarg = scan;
            scan = NULL;
        } else {
            optarg = argv[optind];
            optind++;
        }
    }
```

```
    return(c);
}
```

Sockets LAN Programming Using Java

In this example the Java program connects to the signal generator via sockets LAN. This program requires Java version 1.1 or later be installed on your PC. To run the program perform the following steps:

1. In the code example below, type in the hostname or IP address of your signal generator. For example, `String instrumentName = (your signal generator's hostname).`
2. Copy the program as `ScpiSockTest.java` and save it in a convenient directory on your computer. For example save the file to the `C:\jdk1.3.0_2\bin\javac` directory.
3. Launch the Command Prompt program on your computer. Click **Start** > **Programs** > **Command Prompt**.
4. Compile the program. At the command prompt type: `javac ScpiSockTest.java.` The directory path for the Java compiler must be specified. For example: `C:\>jdk1.3.0_02\bin\javac ScpiSockTest.java`
5. Run the program by typing `java ScpiSockTest` at the command prompt.
6. Type `exit` at the command prompt to end the program.

Generating a CW Signal Using Java

The following program example is available on the signal generator Documentation CD-ROM as `javaex.txt`.

```
//**************************************************************************
// PROGRAM NAME: javaex.txt                                                 // Sample java
program to talk to the signal generator via SCPI-over-sockets
// This program requires Java version 1.1 or later.
// Save this code as ScpiSockTest.java
// Compile by typing: javac ScpiSockTest.java
// Run by typing: java ScpiSockTest
// The signal generator is set for 1 GHz and queried for its id string
//**************************************************************************

import java.io.*;
import java.net.*;
class ScpiSockTest
{
    public static void main(String[] args)
    {
       String instrumentName = "xxxxx";         // Put instrument hostname here
try
        {
```

```
        Socket t = new Socket(instrumentName,5025);  // Connect to instrument
                                                                  // Setup read/write mechanism
             BufferedWriter out =
             new BufferedWriter(
             new OutputStreamWriter(t.getOutputStream()));
             BufferedReader in =
             new BufferedReader(
             new InputStreamReader(t.getInputStream()));
             System.out.println("Setting frequency to 1 GHz...");
             out.write("freq 1GHz\n");               // Sets frequency
             out.flush();
             System.out.println("Waiting for source to settle...");
             out.write("*opc?\n");                   // Waits for completion
             out.flush();
             String opcResponse = in.readLine();
             if (!opcResponse.equals("1"))
              {
               System.err.println("Invalid response to '*OPC?'!");
               System.exit(1);
              }
        System.out.println("Retrieving instrument ID...");
        out.write("*idn?\n");                      // Querys the id string
        out.flush();
        String idnResponse = in.readLine();        // Reads the id string
                                                   // Prints the id string
        System.out.println("Instrument ID: " + idnResponse);
        }
        catch (IOException e)
        {
        System.out.println("Error" + e);
      }
    }
}
```

Sockets LAN Programming Using PERL

This example uses PERL to control the signal generator over the sockets LAN interface. The signal generator frequency is set to 1 GHz, queried for operation complete and then queried for it's identify string. This example was developed using PERL version 5.6.0 and requires a PERL version with the IO::Socket library.

1. In the code below, enter your signal generator's hostname in place of the `xxxxx` in the code line: `my $instrumentName= "xxxxx";` .
2. Save the code listed below using the filename `lanperl`.
3. Run the program by typing `perl lanperl` at the UNIX term window prompt.

Setting the Power Level and Sending Queries Using PERL

The following program example is available on the signal generator Documentation CD-ROM as `perl.txt`.

```
#!/usr/bin/perl
# PROGRAM NAME: perl.txt
# Example of talking to the signal generator via SCPI-over-sockets
#
use IO::Socket;
# Change to your instrument's hostname
my $instrumentName = "xxxxx";

# Get socket
$sock = new IO::Socket::INET ( PeerAddr => $instrumentName,
                               PeerPort => 5025,
                               Proto => 'tcp',
                               );
die "Socket Could not be created, Reason: $!\n" unless $sock;

# Set freq
print "Setting frequency...\n";
print $sock "freq 1 GHz\n";

# Wait for completion
print "Waiting for source to settle...\n";
print $sock "*opc?\n";
my $response = <$sock>;
chomp $response;           # Removes newline from response
if ($response ne "1")
{
   die "Bad response to '*OPC?' from instrument!\n";
}
```

```
# Send identification query
print $sock "*IDN?\n";
$response = <$sock>;
chomp $response;
print "Instrument ID: $response\n";
```

RS-232 Programming Interface Examples (ESG/PSG/E8663B Only)

- "Interface Check Using HP BASIC" on page 142
- "Interface Check Using VISA and C" on page 143
- "Queries Using HP Basic and RS-232" on page 145
- "Queries for RS-232 Using VISA and C" on page 146

Before Using the Examples

Before using the examples: On the signal generator select the following settings:

- Baud Rate - 9600 must match computer's baud rate
- RS-232 Echo - Off

Use an RS-232 cable, that is compatible with Table 2-2 on page 61.

Interface Check Using HP BASIC

This example program causes the signal generator to perform an instrument reset. The SCPI command `*RST` will place the signal generator into a pre-defined state.

The serial interface address for the signal generator in this example is 9. The serial port used is COM1 (Serial A on some computers). Refer to "Using RS–232 (ESG, PSG, and E8663B Only)" on page 58 for more information.

Watch for the signal generator's Listen annunciator (`L`) and the 'remote preset....' message on the front panel display. If there is no indication, check that the RS-232 cable is properly connected to the computer serial port and that the manual setup listed above is correct.

If the compiler displays an error message, or the program hangs, it is possible that the program was typed incorrectly. Press the signal generator's **Reset RS-232** softkey and re-run the program. Refer to "If You Have Problems" on page 63 for more help.

The following program example is available on the signal generator's Documentation CD-ROM as `rs232ex1.txt`.

```
10    !*******************************************************************************
20    !
30    !  PROGRAM NAME:         rs232ex1.txt
40    !
50    !  PROGRAM DESCRIPTION:  This program verifies that the RS-232 connections and
60    !                        interface are functional.
70    !
80    !  Connect the UNIX workstation to the signal generator using an RS-232 cable
90    !
100   !
```

```
110   !  Run HP BASIC, type in the following commands and then RUN the program
120   !
130   !
140   !*************************************************************************
150   !
160    INTEGER Num
170    CONTROL 9,0;1       ! Resets the RS-232 interface
180    CONTROL 9,3;9600    ! Sets the baud rate to match the sig gen
190    STATUS 9,4;Stat     ! Reads the value of register 4
200    Num=BINAND(Stat,7)  ! Gets the AND value
210    CONTROL 9,4;Num     ! Sets parity to NONE
220    OUTPUT 9;"*RST"     ! Outputs reset to the sig gen
230    END                 ! End the program
```

Interface Check Using VISA and C

This program uses VISA library functions to communicate with the signal generator. The program verifies that the RS-232 connections and interface are functional. In this example the COM2 port is used. The serial port is referred to in the VISA library as 'ASRL1' or 'ASRL2' depending on the computer serial port you are using. Launch Microsoft Visual C++, add the required files, and enter the following code into the .cpp source file. `rs232ex1.cpp` performs the following functions:

- prompts the user to set the power on the signal generator to 0 dBm
- error checking
- resets the signal generator to power level of –135 dBm

The following program example is available on the signal generator Documentation CD-ROM as `rs232ex1.cpp`.

```
//*****************************************************************************
// PROGRAM NAME:        rs232ex1.cpp
//
// PROGRAM DESCRIPTION: This code example uses the RS-232 serial interface to
// control the signal generator.
//
// Connect the computer to the signal generator using an RS-232 serial cable.
// The user is asked to set the signal generator for a 0 dBm power level
// A reset command *RST is sent to the signal generator via the RS-232
// interface and the power level will reset to the -135 dBm level.The default
// attributes e.g. 9600 baud, no parity, 8 data bits,1 stop bit are used.
// These attributes can be changed using VISA functions.
//
// IMPORTANT: Set the signal generator BAUD rate to 9600 for this test
//*****************************************************************************

#include <visa.h>
#include <stdio.h>
```

```
#include "StdAfx.h"
#include <stdlib.h>
#include <conio.h>



void main ()
{

int baud=9600;// Set baud rate to 9600
printf("Manually set the signal generator power level to 0 dBm\n");
printf("\n");
printf("Press any key to continue\n");
getch();
printf("\n");
ViSession defaultRM, vi;// Declares a variable of type ViSession
// for instrument communication on COM 2 port
ViStatus viStatus = 0;
                        // Opens session to RS-232 device at serial port 2
viStatus=viOpenDefaultRM(&defaultRM);
viStatus=viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &vi);

if(viStatus){// If operation fails, prompt user
    printf("Could not open ViSession!\n");
    printf("Check instruments and connections\n");
    printf("\n");
    exit(0);}
// initialize device
viStatus=viEnableEvent(vi, VI_EVENT_IO_COMPLETION, VI_QUEUE,VI_NULL);

viClear(vi);// Sends device clear command
// Set attributes for the session
viSetAttribute(vi,VI_ATTR_ASRL_BAUD,baud);
viSetAttribute(vi,VI_ATTR_ASRL_DATA_BITS,8);

viPrintf(vi, "*RST\n");// Resets the signal generator
printf("The signal generator has been reset\n");
printf("Power level should be -135 dBm\n");
printf("\n");// Prints new line character to the display
viClose(vi);// Closes session
viClose(defaultRM);// Closes default session
}
```

Queries Using HP Basic and RS-232

This example program demonstrates signal generator query commands over RS-232. Query commands are of the type *IDN? and are identified by the question mark that follows the mnemonic. rs232ex2.txt performs the following functions:

- resets the RS-232 interface
- sets the baud rate to match the signal generator rate
- reads the value of register 4
- queries the signal generator ID
- sets and queries the power level

Start HP Basic, type in the following commands, and then RUN the program:

The following program example is available on the signal generator Documentation CD-ROM as rs232ex2.txt.

```
10    !*******************************************************************************
20    !
30    !  PROGRAM NAME:         rs232ex2.txt
40    !
50    !  PROGRAM DESCRIPTION:  In this example, query commands are used to read
60    !                        data from the signal generator.
70    !
80    !  Start HP Basic, type in the following code and then RUN the program.
90    !
100   !*******************************************************************************
110   !
120    INTEGER Num
130    DIM Str$[200],Str1$[20]
140    CONTROL 9,0;1              ! Resets the RS-232 interface
150    CONTROL 9,3;9600           ! Sets the baud rate to match signal generator rate
160    STATUS 9,4;Stat            ! Reads the value of register 4
170    Num=BINAND(Stat,7)         ! Gets the AND value
180    CONTROL 9,4;Num            ! Sets the parity to NONE
190    OUTPUT 9;"*IDN?"           ! Querys the sig gen ID
200    ENTER 9;Str$               ! Reads the ID
210    WAIT 2                     ! Waits 2 seconds
220    PRINT "ID =",Str$          ! Prints ID to the screen
230    OUTPUT 9;"POW:AMPL -5 dbm" ! Sets the the power level to -5 dbm
240    OUTPUT 9;"POW?"            ! Querys the power level of the sig gen
250    ENTER 9;Str1$              ! Reads the queried value
260    PRINT "Power = ",Str1$     ! Prints the power level to the screen
270    END                        ! End the program
```

Queries for RS-232 Using VISA and C

This example uses VISA library functions to communicate with the signal generator. The program verifies that the RS-232 connections and interface are functional. Launch Microsoft Visual C++, add the required files, and enter the following code into your .cpp source file. `rs232ex2.cpp` performs the following functions:

- error checking
- reads the signal generator response
- flushes the read buffer
- queries the signal generator for power
- reads the signal generator power

The following program example is available on the signal generator Documentation CD-ROM as `rs232ex2.cpp`.

```
//*****************************************************************************
//
// PROGRAM NAME:       rs232ex2.cpp
//
// PROGRAM DESCRIPTION: This code example uses the RS-232 serial interface to control
// the signal generator.
//
// Connect the computer to the signal generator using the RS-232 serial cable
// and enter the following code into the project .cpp source file.
// The program queries the signal generator ID string and sets and queries the power
// level. Query results are printed to the screen. The default attributes e.g. 9600 baud,
// parity, 8 data bits,1 stop bit are used. These attributes can be changed using VISA
// functions.
//
// IMPORTANT: Set the signal generator BAUD rate to 9600 for this test
//*****************************************************************************


#include <visa.h>
#include <stdio.h>
#include "StdAfx.h"
#include <stdlib.h>
#include <conio.h>

#define MAX_COUNT 200

int main (void)
{

ViStatusstatus;  // Declares a type ViStatus variable
```

```
ViSessiondefaultRM, instr;// Declares type ViSession variables
ViUInt32retCount;  // Return count for string I/O
ViCharbuffer[MAX_COUNT];// Buffer for string I/O

status = viOpenDefaultRM(&defaultRM);// Initializes the system
// Open communication with Serial Port 2
status = viOpen(defaultRM, "ASRL2::INSTR", VI_NULL, VI_NULL, &instr);

if(status){// If problems, then prompt user
printf("Could not open ViSession!\n");
printf("Check instruments and connections\n");
printf("\n");
exit(0);}
                                        // Set timeout for 5 seconds
viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);
// Asks for sig gen ID string
 status = viWrite(instr, (ViBuf)"*IDN?\n", 6, &retCount);

// Reads the sig gen response
 status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';// Indicates the end of the string
printf("Signal Generator ID: "); // Prints header for ID
printf(buffer);// Prints the ID string to the screen
printf("\n");// Prints carriage return
// Flush the read buffer
// Sets sig gen power to -5dbm
status = viWrite(instr, (ViBuf)"POW:AMPL -5dbm\n", 15, &retCount);
// Querys the sig gen for power level
status = viWrite(instr, (ViBuf)"POW?\n",5,&retCount);
// Read the power level
status = viRead(instr, (ViBuf)buffer, MAX_COUNT, &retCount);
buffer[retCount]= '\0';// Indicates the end of the string
printf("Power level = ");// Prints header to the screen
printf(buffer);// Prints the queried power level
printf("\n");
status = viClose(instr);// Close down the system
status = viClose(defaultRM);
return 0;
}
```

4 Programming the Status Register System

This chapter provides the following major sections:

- Overview on page 149
- Status Register Bit Values on page 160
- Accessing Status Register Information on page 161
- Status Byte Group on page 167
- Status Groups on page 169

Overview

NOTE Some of the status bits and register groups only apply to select signal generators with certain options do not apply to the E8663B. For more specific information on each exception, refer to the following:

- Standard Operation Condition Register bits (see Table 4-6 on page 173)
- Baseband Operation Status Group (see page 175)
- Data Questionable Condition Register bits (see Table 4-8 on page 179)
- Data Questionable Power Condition Register bits (see Table 4-9 on page 182)
- Data Questionable Frequency Condition Register bits (see Table 4-10 on page 185)
- Data Questionable Modulation Condition Register bits (see Table 4-11 on page 188)
- Data Questionable Calibration Condition Register bit (see Table 4-12 on page 191)
- Data Questionable Bert Status Group (see page 193)

During remote operation, you may need to monitor the status of the signal generator for error conditions or status changes. You can use the signal generator's status register system to monitor error conditions, or condition changes, or both. In general, the error queue is easier to use than the status registers, but the status registers provide some additional information not found in the error queue. For more information on using the signal generator's SCPI commands to query the signal generator's error queue, refer to the signal generator's SCPI command reference guide, to see if any errors have occurred.

The signal generator's status register system provides two major advantages:

- You can monitor the settling of the signal generator using the settling bit of the Standard Operation Status Group's condition register.
- You can use the service request (SRQ) interrupt technique to avoid status polling, therefore giving a speed advantage.

The signal generator's instrument status system provides complete SCPI compliant data structures for reporting instrument status using the register model.

The SCPI register model of the status system has multiple registers that are arranged in a hierarchical order. The lower-priority status registers propagate their data to the higher-priority

registers using summary bits. The Status Byte Register is at the top of the hierarchy and contains the status information for lower level registers. The lower level registers monitor specific events or conditions.

The lower level status registers are grouped according to their functionality. For example, the Data Questionable Frequency Status Group consists of five registers. This chapter may refer to a group as a register so that the cumbersome longer description is avoided. For example, the Standard Operation Status Group's Condition Register can be referred to as the Standard Operation Status register. Refer to "Status Groups" on page 169 for more information.

Figure 4-1, Figure 4-2, Figure 4-3, and Figure 4-4, Figure 4-5, Figure 4-6, Figure 4-7, and Figure 4-8 shows each signal generator model's the signal generator's status byte register system and hierarchy.

The status register systems use IEEE 488.2 commands (those beginning with *) to access the higher-level summary registers (refer to the *SCPI Reference*). Access Lower-level registers by using STATus commands.

Overall Status Byte Register Systems

- "N5161A/62A/81A/82A/83A: Overall Status Byte Register System (1 of 2)" on page 152
- "N5 161A/62A/81A/82A/83A: Overall Status Byte Register System (2 of 2)" on page 153
- "E8663B: Overall Status Byte Register System (1 of 2)" on page 154
- "E8663B: Overall Status Byte Register System (2 of 2)" on page 155
- "E4428C/38C: Overall Status Byte Register System (1 of 2)" on page 156
- "E4428C/38C: Overall Status Byte Register System (2 of 2)" on page 157
- "E8257D/67D: Overall Status Byte Register System (1 of 2)" on page 158
- "E8257D/67D: Overall Status Byte Register System (2 of 2)" on page 159

Figure 4-1 **N5161A/62A/81A/82A/83A: Overall Status Byte Register System (1 of 2)**

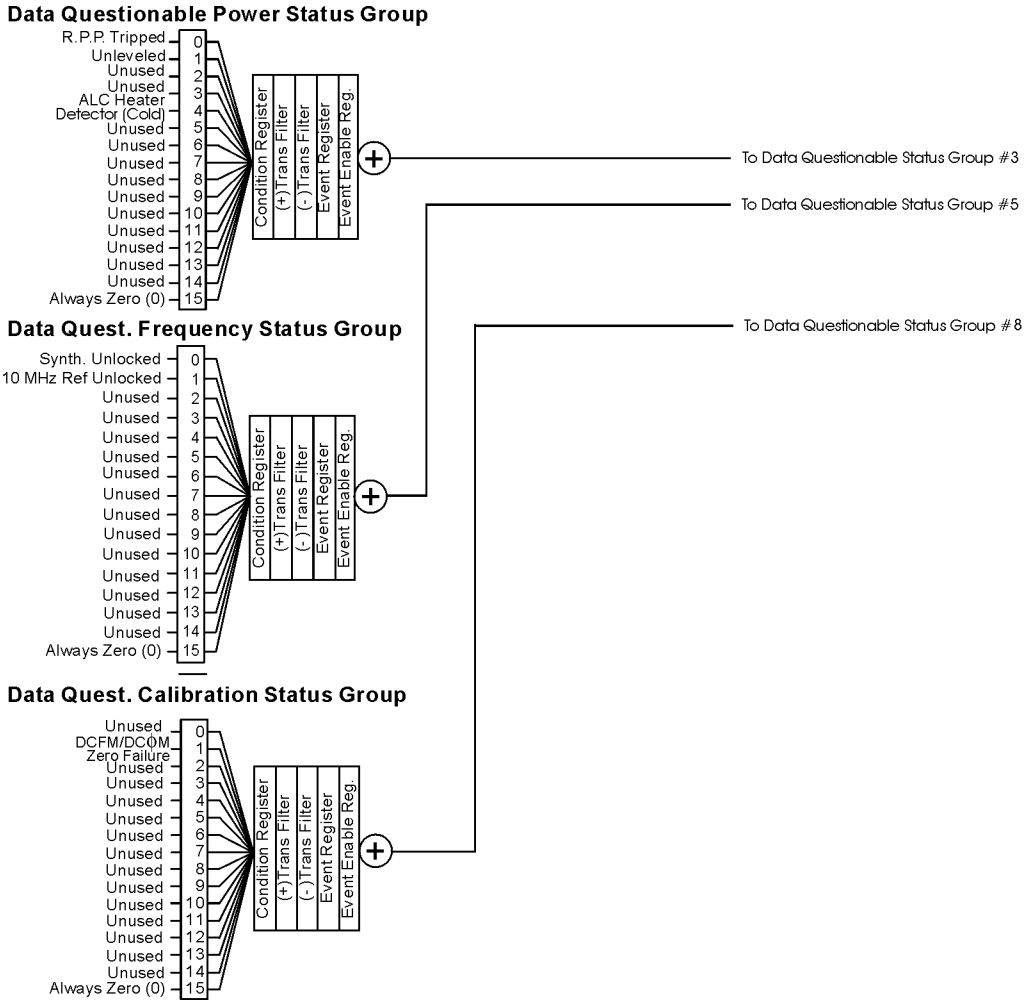


Figure 4-2 **N5161A/62A/81A/82A/83A: Overall Status Byte Register System (2 of 2)**

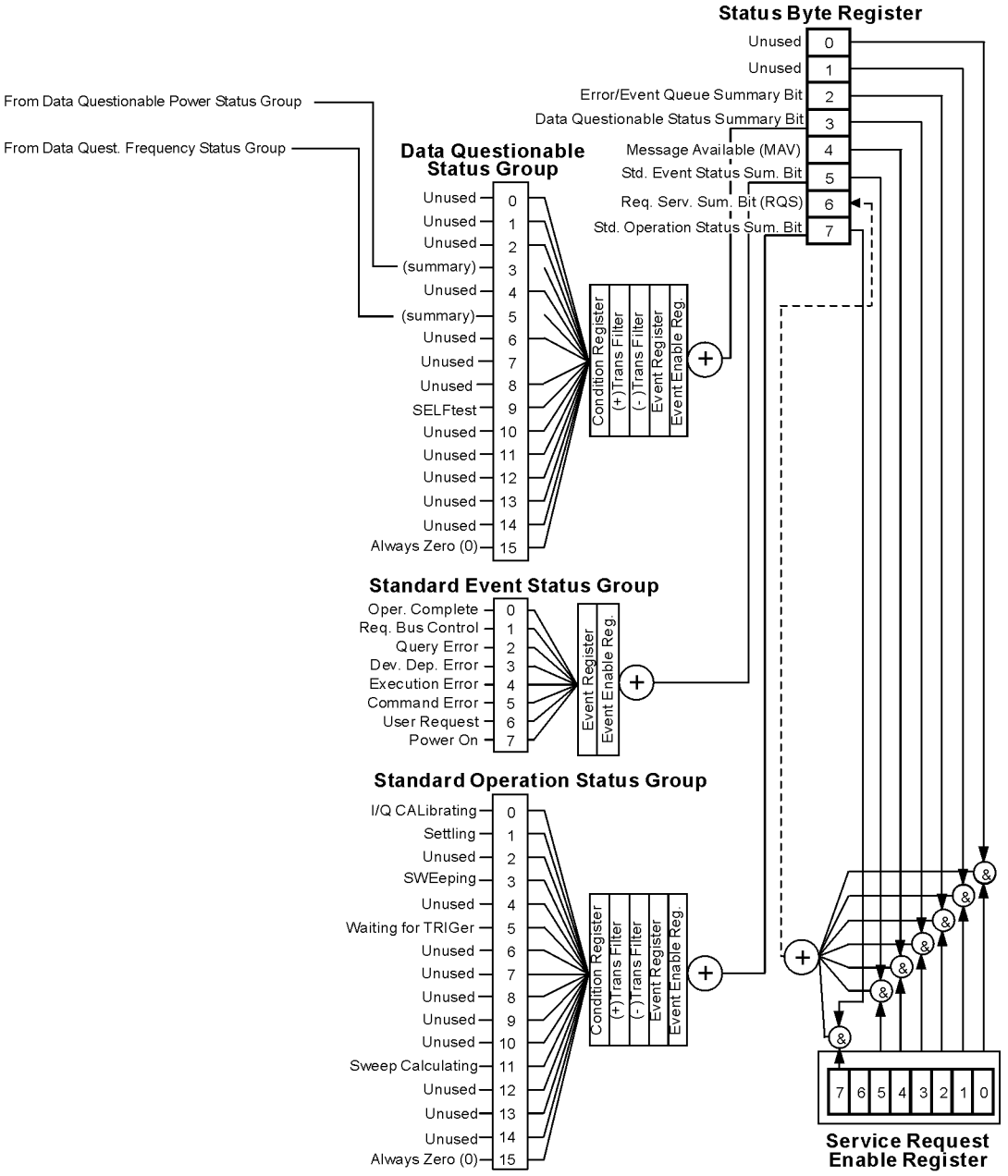


Figure 4-3 E8663B: Overall Status Byte Register System (1 of 2)

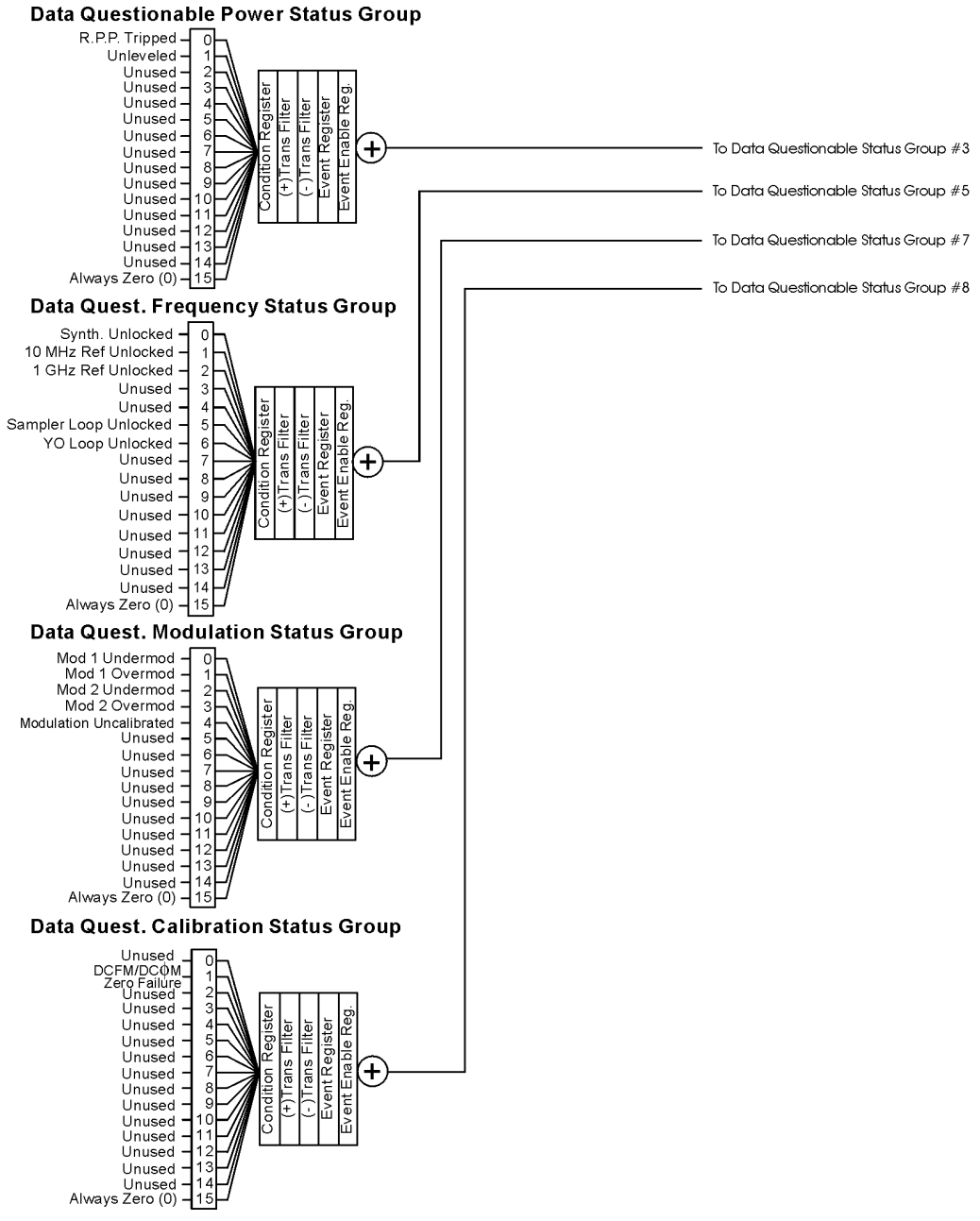


Figure 4-4 **E8663B: Overall Status Byte Register System (2 of 2)**

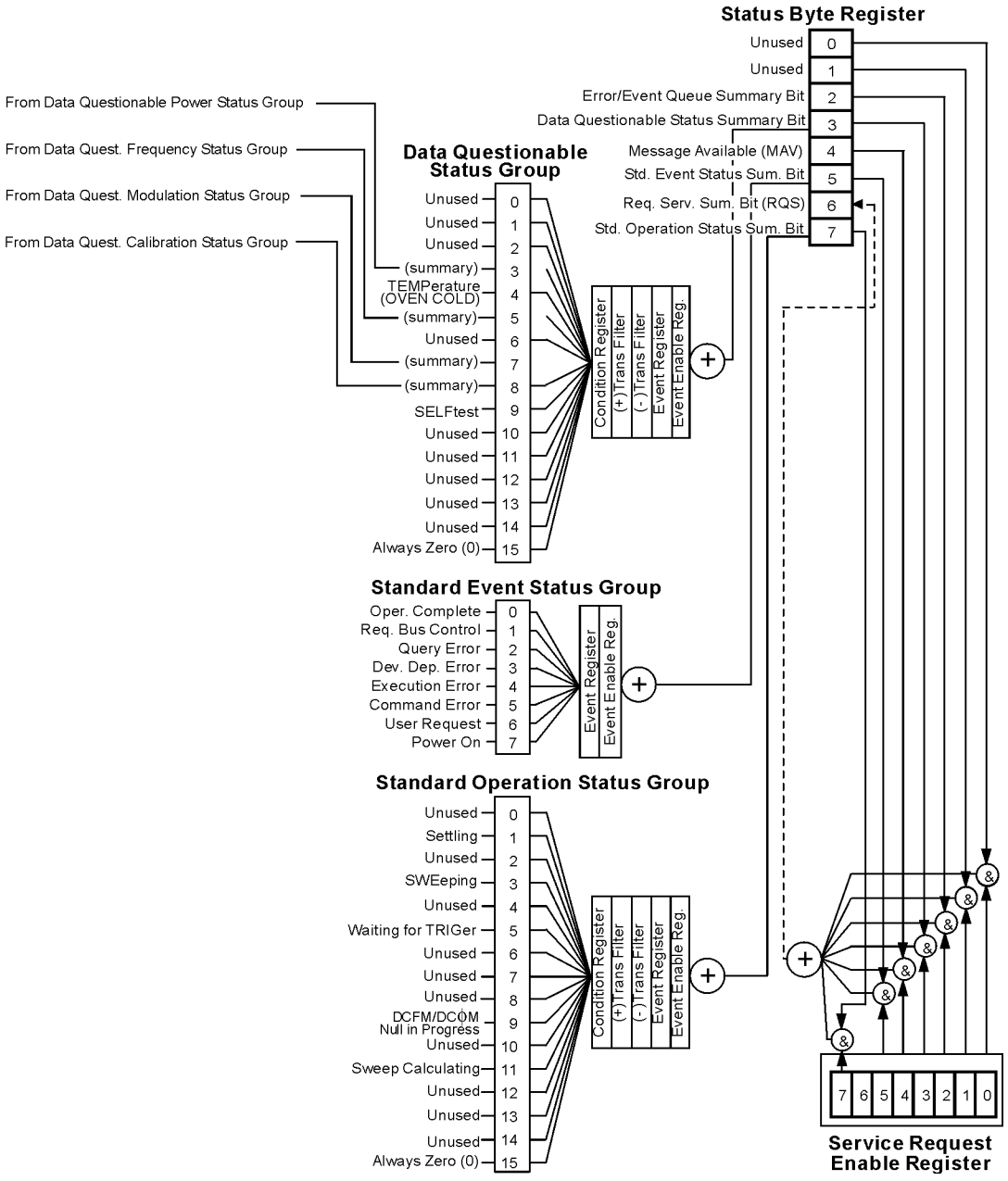


Figure 4-5 E4428C/38C: Overall Status Byte Register System (1 of 2)

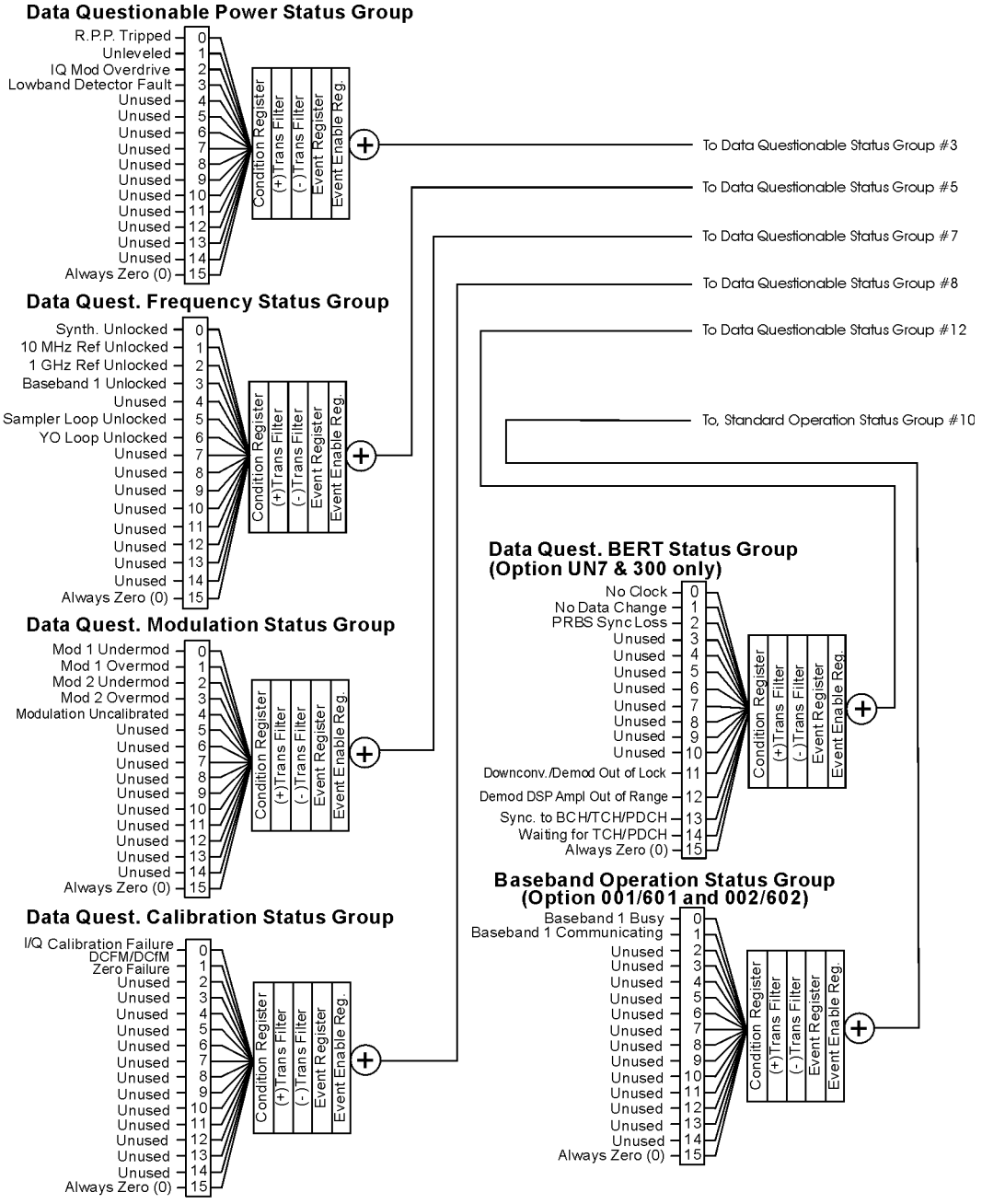


Figure 4-6 **E4428C/38C: Overall Status Byte Register System (2 of 2)**

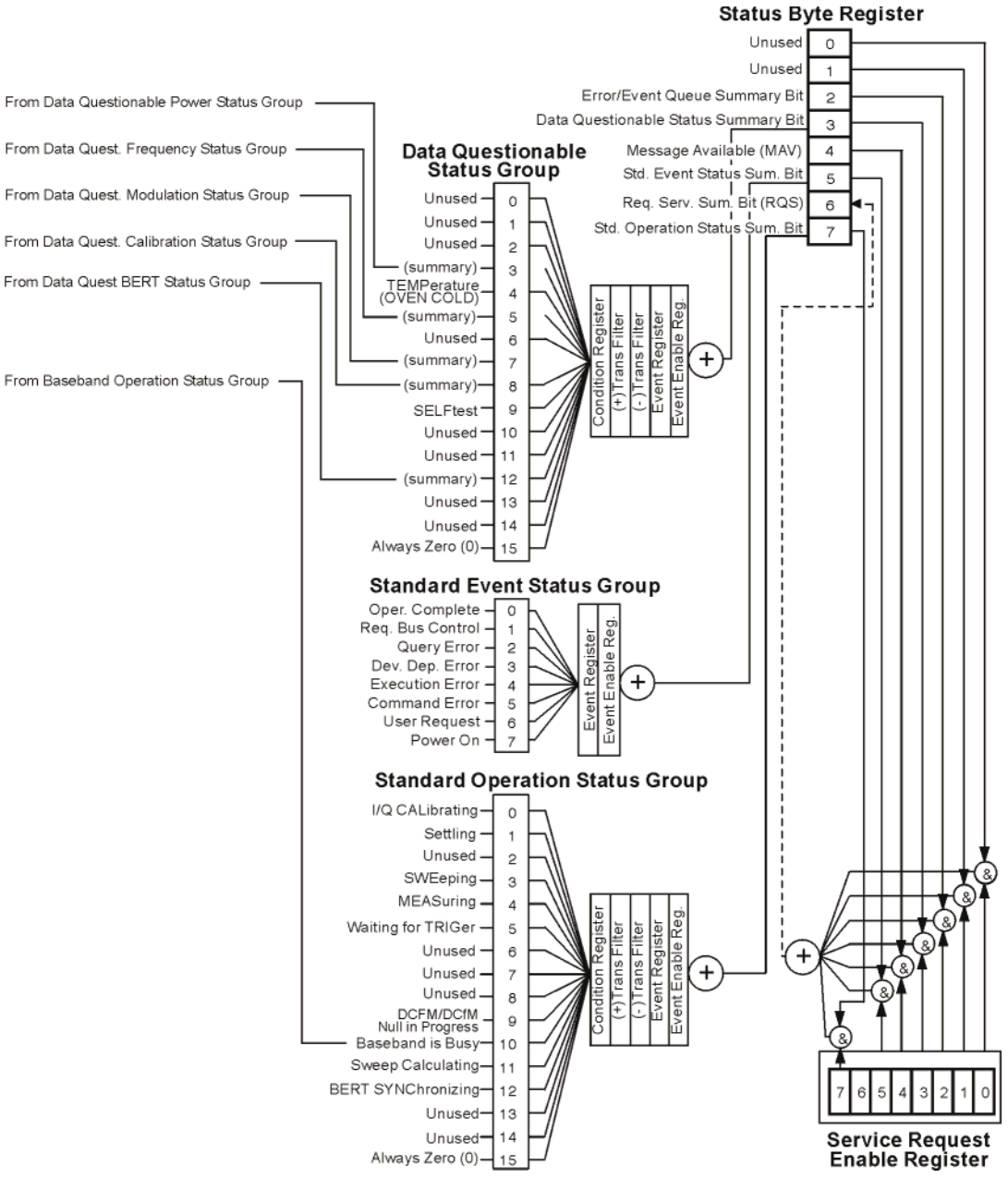


Figure 4-7 E8257D/67D: Overall Status Byte Register System (1 of 2)

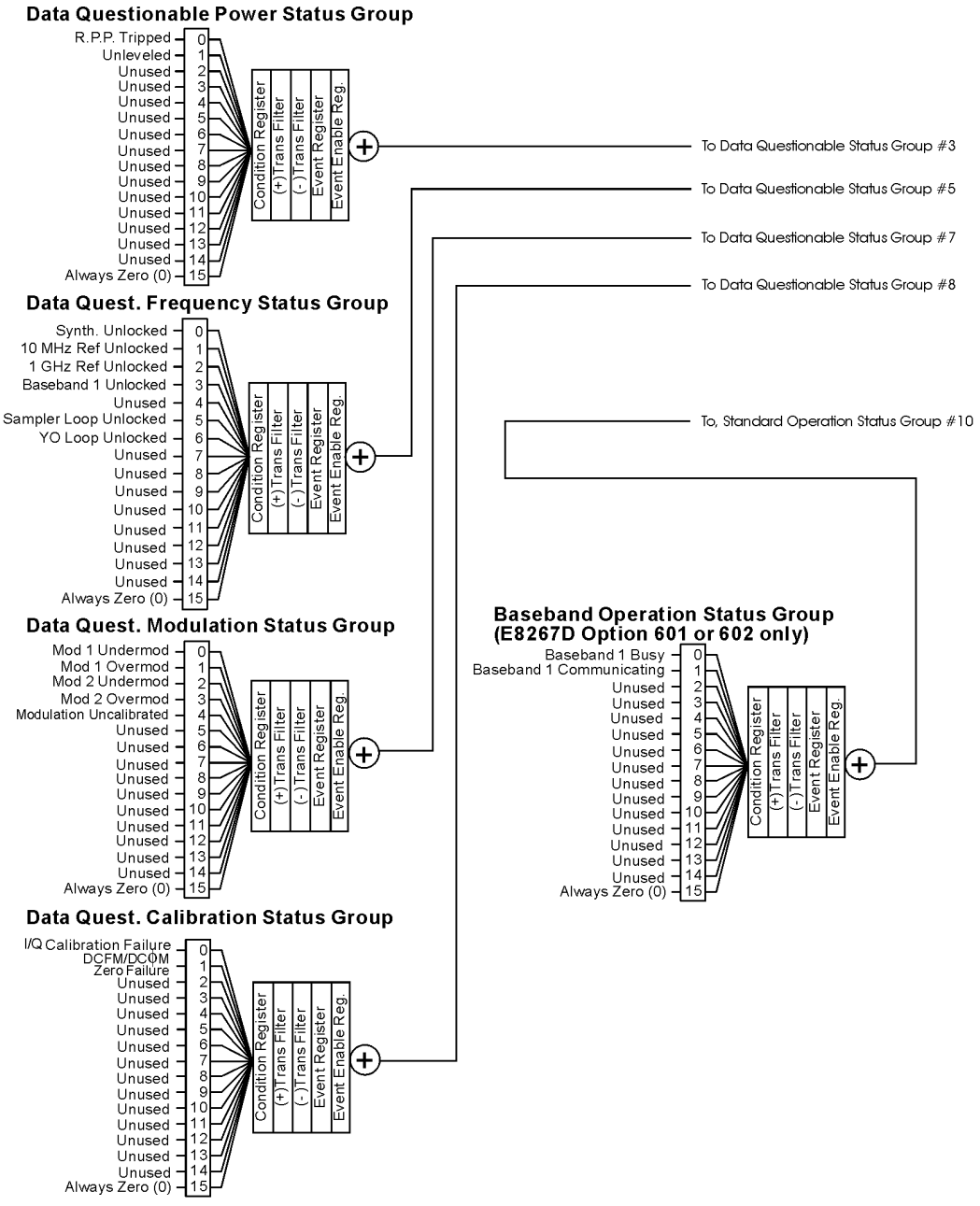
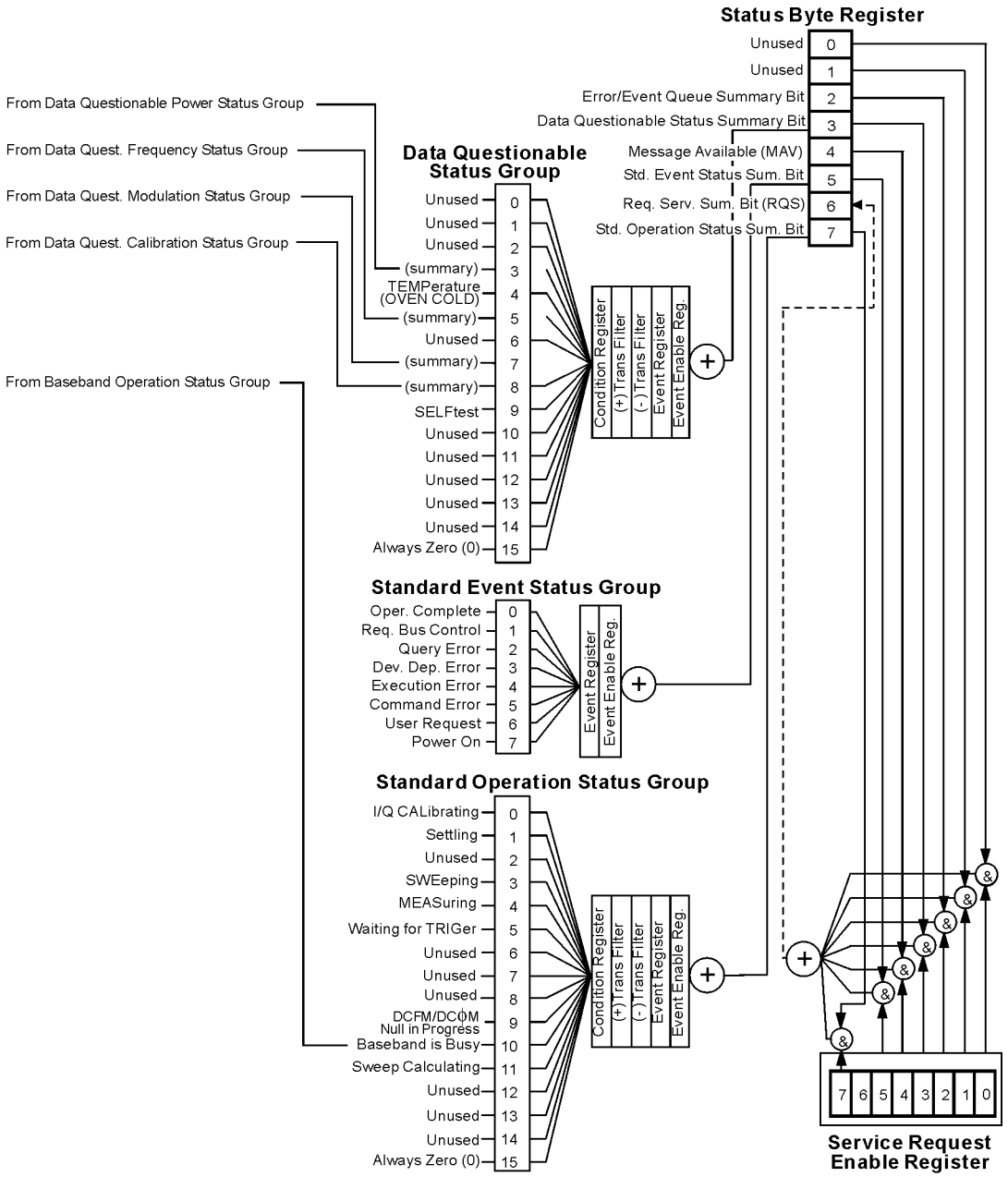


Figure 4-8 **E8257D/67D: Overall Status Byte Register System (2 of 2)**



Status Register Bit Values

Each bit in a register is represented by a decimal value based on its location in the register (see Table 4-1).

- To enable a particular bit in a register, send its value with the SCPI command. Refer to the signal generator's SCPI command listing for more information.
- To enable more than one bit, send the sum of all the bits that you want to enable.
- To verify the bits set in a register, query the register.

Example: Enable a Register

To enable bit 0 and bit 6 of the Standard Event Status Group's Event Register:

1. Add the decimal value of bit 0 (1) and the decimal value of bit 6 (64) to give a decimal value of 65.
2. Send the sum with the command: `*ESE 65`.

Example: Query a Register

To query a register for a condition, send a SCPI query command. For example, if you want to query the Standard Operation Status Group's Condition Register, send the command:

STATus:OPERation:CONDition?

If bit 7, bit 3 and bit 2 in this register are set (bits = 1) then the query will return the decimal value 140. The value represents the decimal values of bit 7, bit 3 and bit 2: 128 + 8 + 4 = 140.

Table 4-1 Status Register Bit Decimal Values

Decimal Value	Always 0	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
Bit Number	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

NOTE Bit 15 is not used and is always set to zero.

Accessing Status Register Information

1. Determine which register contains the bit that reports the condition. Refer to Figure 4-1 on page 152 through Figure 4-8 on page 159 for register location and names.
2. Send the unique SCPI query that reads that register.
3. Examine the bit to see if the condition has changed.

Determining What to Monitor

You can monitor the following conditions:

- current signal generator hardware and firmware status
- whether a particular condition (bit) has occurred

Monitoring Current Signal Generator Hardware and Firmware Status

To monitor the signal generator's operating status, you can query the condition registers. These registers represent the current state of the signal generator and are updated in real time. When the condition monitored by a particular bit becomes true, the bit sets to 1. When the condition becomes false, the bit resets to 0.

Monitoring Whether a Condition (Bit) has Changed

The transition registers determine which bit transition (condition change) should be recorded as an event. The transitions can be positive to negative, negative to positive, or both. To monitor a certain condition, enable the bit associated with the condition in the associated positive and negative registers.

Once you have enabled a bit via the transition registers, the signal generator monitors it for a change in its condition. If this change in condition occurs, the corresponding bit in the event register will be set to 1. When a bit becomes true (set to 1) in the event register, it stays set until the event register is read or is cleared. You can thus query the event register for a condition even if that condition no longer exists.

To clear the event register, query its contents or send the *CLS command, which clears *all* event registers.

Monitoring When a Condition (Bit) Changes

Once you enable a bit, the signal generator monitors it for a change in its condition. The transition registers are preset to register positive transitions (a change going from 0 to 1). This can be changed so the selected bit is detected if it goes from true to false (negative transition), or if either transition occurs.

Deciding How to Monitor

You can use either of two methods described below to access the information in status registers (both methods allow you to monitor one or more conditions).

- **The polling method**

  In the polling method, the signal generator has a passive role. It tells the controller that conditions have changed only when the controller asks the right question. This is accomplished by a program loop that continually sends a query.

The polling method works well if you do not need to know about changes the moment they occur. Use polling in the following situations:

- when you use a programming language/development environment or IO interface that does not support SRQ interrupts
- when you want to write a simple, single-purpose program and don't want the added complexity of setting up an SRQ handler

- **The service request (SRQ) method**

  In the SRQ method (described in the following section), the signal generator takes a more active role. It tells the controller when there has been a condition change without the controller asking. Use the SRQ method to detect changes using the polling method, where the program must repeatedly read the registers.

  Use the SRQ method if you must know immediately when a condition changes. Use the SRQ method in the following situations:

  - when you need time-critical notification of changes
  - when you are monitoring more than one device that supports SRQs
  - when you need to have the controller do something else while waiting
  - when you can't afford the performance penalty inherent to polling

Using the Service Request (SRQ) Method

The programming language, I/O interface, and programming environment must support SRQ interrupts (for example: BASIC or VISA used with GPIB and VXI-11 over the LAN). Using this method, you must do the following:

1. Determine which bit monitors the condition.
2. Send commands to enable the bit that monitors the condition (transition registers).
3. Send commands to enable the summary bits that report the condition (event enable registers).
4. Send commands to enable the status byte register to monitor the condition.
5. Enable the controller to respond to service requests.

The controller responds to the SRQ as soon as it occurs. As a result, the time the controller would otherwise have used to monitor the condition, as in a loop method, can be used to perform other tasks. The application determines how the controller responds to the SRQ.

When a condition changes and that condition has been enabled, the request service summary (RQS) bit in the status byte register is set. In order for the controller to respond to the change, the Service Request Enable Register needs to be enabled for the bit(s) that will trigger the SRQ.

Generating a Service Request

The Service Request Enable Register lets you choose the bits in the Status Byte Register that will trigger a service request. Send the *SRE <num> command where <num> is the sum of the decimal values of the bits you want to enable.

For example, to enable bit 7 on the Status Byte Register (so that whenever the Standard Operation Status register summary bit is set to 1, a service request is generated) send the command *SRE 128.

Refer to Figure 4-1 on page 152 through Figure 4-8 on page 159 for bit positions and values.

The query command *SRE? returns the decimal value of the sum of the bits previously enabled with the *SRE <num> command.

To query the Status Byte Register, send the command *STB?. The response will be the decimal sum of the bits which are set to 1. For example, if bit 7 and bit 3 are set, the decimal sum will be 136 (bit 7 = 128 and bit 3 = 8).

NOTE Multiple Status Byte Register bits can assert an SRQ, however only one bit at a time can set the RQS bit. All bits that are asserting an SRQ will be read as part of the status byte when it is queried or serial polled.

The SRQ process asserts SRQ as true and sets the status byte's RQS bit to 1. Both actions are necessary to inform the controller that the signal generator requires service. Asserting SRQ informs the controller that some device on the bus requires service. Setting the RQS bit allows the controller to determine which signal generator requires service.

This process is initiated if both of the following conditions are true:

- The corresponding bit of the Service Request Enable Register is also set to 1.
- The signal generator does not have a service request pending.

  A service request is considered to be pending between the time the signal generator's SRQ process is initiated and the time the controller reads the status byte register.

If a program enables the controller to detect and respond to service requests, it should instruct the controller to perform a serial poll when SRQ is true. Each device on the bus returns the contents of its status byte register in response to this poll. The device whose request service summary (RQS) bit is set to 1 is the device that requested service.

NOTE When you read the signal generator's Status Byte Register with a serial poll, the RQS bit is reset to 0. Other bits in the register are not affected.

If the status register is configured to SRQ on end-of-sweep or measurement and the mode set to continuous, restarting the measurement (INIT command) can cause the measuring bit to pulse low. This causes an SRQ when you have not actually reached the "end-of-sweep" or measurement condition. To avoid this, do the following:

1. Send the command `INITiate:CONTinuous OFF`.
2. Set/enable the status registers.
3. Restart the measurement (send INIT).

Status Register SCPI Commands

Most monitoring of signal generator conditions is done at the highest level using the IEEE 488.2 common commands listed below. You can set and query individual status registers using the commands in the STATus subsystem.

*CLS (clear status) clears the Status Byte Register by emptying the error queue and clearing all the event registers.

*ESE, *ESE? (event status enable) sets and queries the bits in the Standard Event Enable Register which is part of the Standard Event Status Group.

*ESR? (event status register) queries and clears the Standard Event Status Register which is part of the Standard Event Status Group.

*OPC, *OPC? (operation complete) sets bit #0 in the Standard Event Status Register to 1 when all commands have completed. The query stops any new commands from being processed until the current processing is complete, then returns a 1.

*PSC, *PSC? (power-on state clear) sets the power-on state so that it clears the Service Request Enable Register, the Standard Event Status Enable Register, and device-specific event enable registers at power on. The query returns the flag setting from the *PSC command.

*SRE, *SRE? (service request enable) sets and queries the value of the Service Request Enable Register.

*STB? (status byte) queries the value of the status byte register without erasing its contents.

:STATus:PRESet presets all transition filters, non-IEEE 488.2 enable registers, and error/event queue enable registers. (Refer to Table 4-2.)

Table 4-2 Effects of :STATus:PRESet

Register[a]	Value after :STATus:PRESet
:STATus:OPERation:ENABle	0
:STATus:OPERation:NTRansition	0
:STATus:OPERation:PTRransition	32767
:STATus:OPERation:BASeband:ENABle	0
:STATus:OPERation:BASeband:NTRansition	0
:STATus:OPERation:BASeband:PTRransition	32767
:STATus:QUEStionable:CALibration:ENABle	32767
:STATus:QUEStionable:CALibration:NTRansition	32767
:STATus:QUEStionable:CALibration:PTRansition	32767
:STATus:QUEStionable:ENABle	0
:STATus:QUEStionable:NTRansition	0
:STATus:QUEStionable:PTRansition	32767
:STATus:QUEStionable:FREQuency:ENABle	32767
:STATus:QUEStionable:FREQuency:NTRansition	32767
:STATus:QUEStionable:FREQuency:PTRansition	32767
:STATus:QUEStionable:MODulation:ENABle	32767
:STATus:QUEStionable:MODulation:NTRansition	32767
:STATus:QUEStionable:MODulation:PTRansition	32767
:STATus:QUEStionable:POWer:ENABle	32767
:STATus:QUEStionable:POWer:NTRansition	32767
:STATus:QUEStionable:POWer:PTRansition	32767
:STATus:QUEStionable:BERT:ENABle	32767
:STATus:QUEStionable:BERT:NTRansition	32767
:STATus:QUEStionable:BERT:PTRansition	32767

a. Table reflects :STAT:PRES values for an E4438C with options 001, 002, 601, or 602 and UN7. To determine the registers that apply to your signal generator, refer to Figure 4-1 on page 152 through Figure 4-8 on page 159 and Table 4-4 on page 168 through Table 4-13 on page 194.

Table 4-3 Effects of :STATus:PRESet

Register	Value after :STATus:PRESet
:STATus:OPERation:ENABle	0
:STATus:OPERation:NTRansition	0
:STATus:OPERation:PTRransition	32767
:STATus:QUEStionable:CALibration:ENABle	32767
:STATus:QUEStionable:CALibration:NTRansition	32767
:STATus:QUEStionable:CALibration:PTRansition	32767
:STATus:QUEStionable:ENABle	0
:STATus:QUEStionable:NTRansition	0

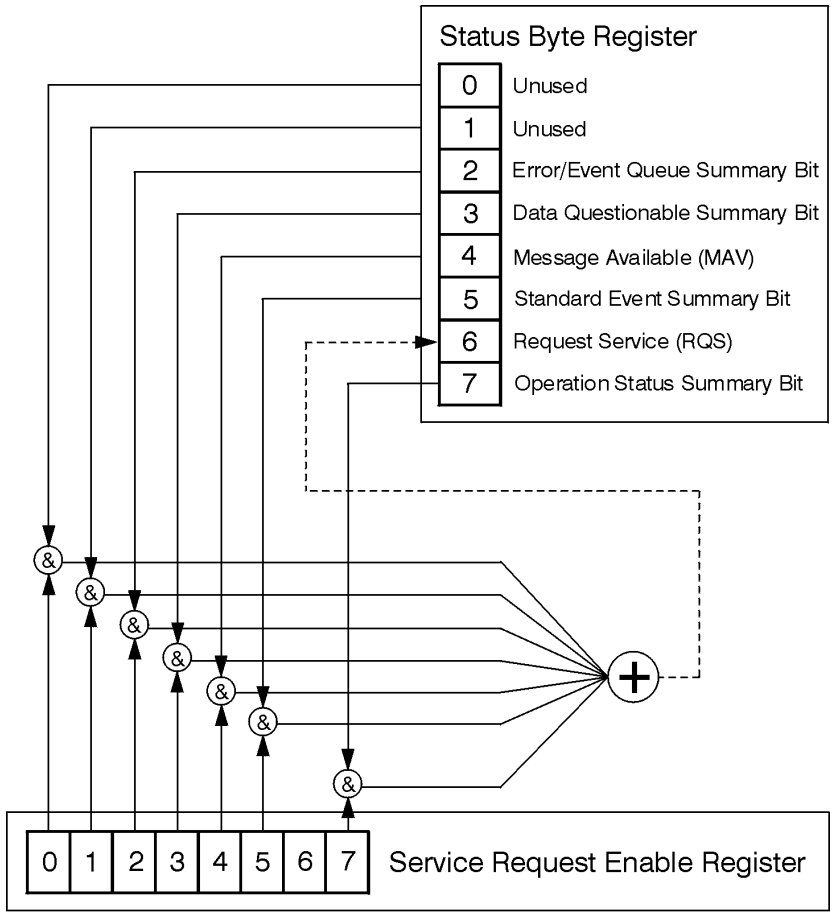
Table 4-3 Effects of :STATus:PRESet

Register	Value after :STATus:PRESet
:STATus:QUEStionable:PTRansition	32767
:STATus:QUEStionable:FREQuency:ENABle	32767
:STATus:QUEStionable:FREQuency:NTRansition	32767
:STATus:QUEStionable:FREQuency:PTRansition	32767
:STATus:QUEStionable:MODulation:ENABle	32767
:STATus:QUEStionable:MODulation:NTRansition	32767
:STATus:QUEStionable:MODulation:PTRansition	32767
:STATus:QUEStionable:POWer:ENABle	32767
:STATus:QUEStionable:POWer:NTRansition	32767
:STATus:QUEStionable:POWer:PTRansition	32767

Status Byte Group

The Status Byte Group includes the Status Byte Register and the Service Request Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



ck721a

Status Byte Register

Table 4-4 Status Byte Register Bits

Bit	Description
0,1	**Unused**. These bits are always set to 0.
2	**Error/Event Queue Summary Bit**. A 1 in this bit position indicates that the SCPI error queue is not empty. The SCPI error queue contains at least one error message.
3	**Data Questionable Status Summary Bit**. A 1 in this bit position indicates that the Data Questionable summary bit has been set. The Data Questionable Event Register can then be read to determine the specific condition that caused this bit to be set.
4	**Message Available**. A 1 in this bit position indicates that the signal generator has data ready in the output queue. There are no lower status groups that provide input to this bit.
5	**Standard Event Status Summary Bit**. A 1 in this bit position indicates that the Standard Event summary bit has been set. The Standard Event Status Register can then be read to determine the specific event that caused this bit to be set.
6	**Request Service (RQS) Summary Bit**. A 1 in this bit position indicates that the signal generator has at least one reason to require service. This bit is also called the Master Summary Status bit (MSS). The individual bits in the Status Byte are individually ANDed with their corresponding service request enable register, then each individual bit value is ORed and input to this bit.
7	**Standard Operation Status Summary Bit**. A 1 in this bit position indicates that the Standard Operation Status Group's summary bit has been set. The Standard Operation Event Register can then be read to determine the specific condition that caused this bit to be set.

Query: `*STB?`

Response: The *decimal* sum of the bits set to 1 including the master summary status bit (MSS) bit 6.

Example: The decimal value 136 is returned when the MSS bit is set low (0).

Decimal sum = 128 (bit 7) + 8 (bit 3)

The decimal value 200 is returned when the MSS bit is set high (1).

Decimal sum = 128 (bit 7) + 8 (bit 3) + 64 (MSS bit)

Service Request Enable Register

The Service Request Enable Register lets you choose which bits in the Status Byte Register trigger a service request.

`*SRE <data>` `<data>` is the sum of the decimal values of the bits you want to enable except bit 6. Bit 6 cannot be enabled on this register. Refer to Figure 4-1 on page 152 through Figure 4-8 on page 159.

Example: To enable bits 7 and 5 to trigger a service request when either corresponding status group register summary bit sets to 1, send the command `*SRE 160` (128 + 32).

Query: `*SRE?`

Response: The decimal value of the sum of the bits previously enabled with the `*SRE <data>` command.

Status Groups

The Standard Operation Status Group and the Data Questionable Status Group consist of the registers listed below. The Standard Event Status Group is similar but does *not* have negative or positive transition filters or a condition register.

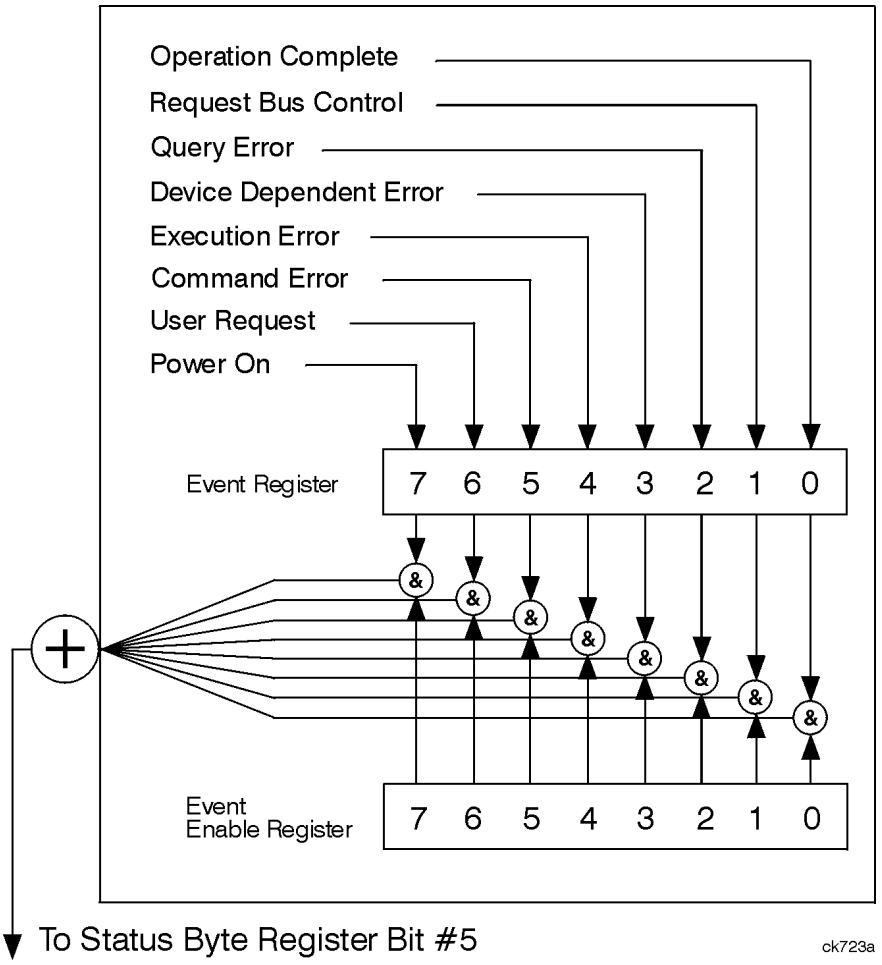
Condition Register	A condition register continuously monitors the hardware and firmware status of the signal generator. There is no latching or buffering for a condition register; it is updated in real time.
Negative Transition Filter	A negative transition filter specifies the bits in the condition register that will set corresponding bits in the event register when the condition bit changes from 1 to 0.
Positive Transition Filter	A positive transition filter specifies the bits in the condition register that will set corresponding bits in the event register when the condition bit changes from 0 to 1.
Event Register	An event register latches transition events from the condition register as specified by the positive and negative transition filters. Once the bits in the event register are set, they remain set until cleared by either querying the register contents or sending the `*CLS` command.
Event Enable Register	An enable register specifies the bits in the event register that generate the summary bit. The signal generator logically ANDs corresponding bits in the event and enable registers and ORs all the resulting bits to produce a summary bit. Summary bits are, in turn, used by the Status Byte Register.

A status group is a set of related registers whose contents are programmed to produce status summary bits. In each status group, corresponding bits in the condition register are filtered by the negative and positive transition filters and stored in the event register. The contents of the event register are logically ANDed with the contents of the enable register and the result is logically ORed to produce a status summary bit in the Status Byte Register.

Standard Event Status Group

The Standard Event Status Group is used to determine the specific event that set bit 5 in the Status Byte Register. This group consists of the Standard Event Status Register (an event register) and the Standard Event Status Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Standard Event Status Register

Table 4-5 Standard Event Status Register Bits

Bit	Description
0	**Operation Complete**. A 1 in this bit position indicates that all pending signal generator operations were completed following execution of the `*OPC` command.
1	**Request Control**. This bit is always set to 0. (The signal generator does not request control.)
2	**Query Error**. A 1 in this bit position indicates that a query error has occurred. Query errors have instrument error numbers from –499 to –400.
3	**Device Dependent Error**. A 1 in this bit position indicates that a device dependent error has occurred. Device dependent errors have instrument error numbers from –399 to –300 and 1 to 32767.
4	**Execution Error**. A 1 in this bit position indicates that an execution error has occurred. Execution errors have instrument error numbers from –299 to –200.
5	**Command Error**. A 1 in this bit position indicates that a command error has occurred. Command errors have instrument error numbers from –199 to –100.
6	**User Request Key (Local)**. A 1 in this bit position indicates that the **Local** key has been pressed. This is true even if the signal generator is in local lockout mode.
7	**Power On**. A 1 in this bit position indicates that the signal generator has been turned off and then on.

Query: `*ESR?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 136 is returned. The decimal sum = 128 (bit 7) + 8 (bit 3).

Standard Event Status Enable Register

The Standard Event Status Enable Register lets you choose which bits in the Standard Event Status Register set the summary bit (bit 5 of the Status Byte Register) to 1.

`*ESE <data>` `<data>` is the sum of the decimal values of the bits you want to enable.

Example: To enable bit 7 and bit 6 so that whenever either of those bits are set to 1, the Standard Event Status summary bit of the Status Byte Register is set to 1. Send the command `*ESE 192` (128 + 64).

Query: `*ESE?`

Response: Decimal value of the sum of the bits previously enabled with the `*ESE <data>` command.

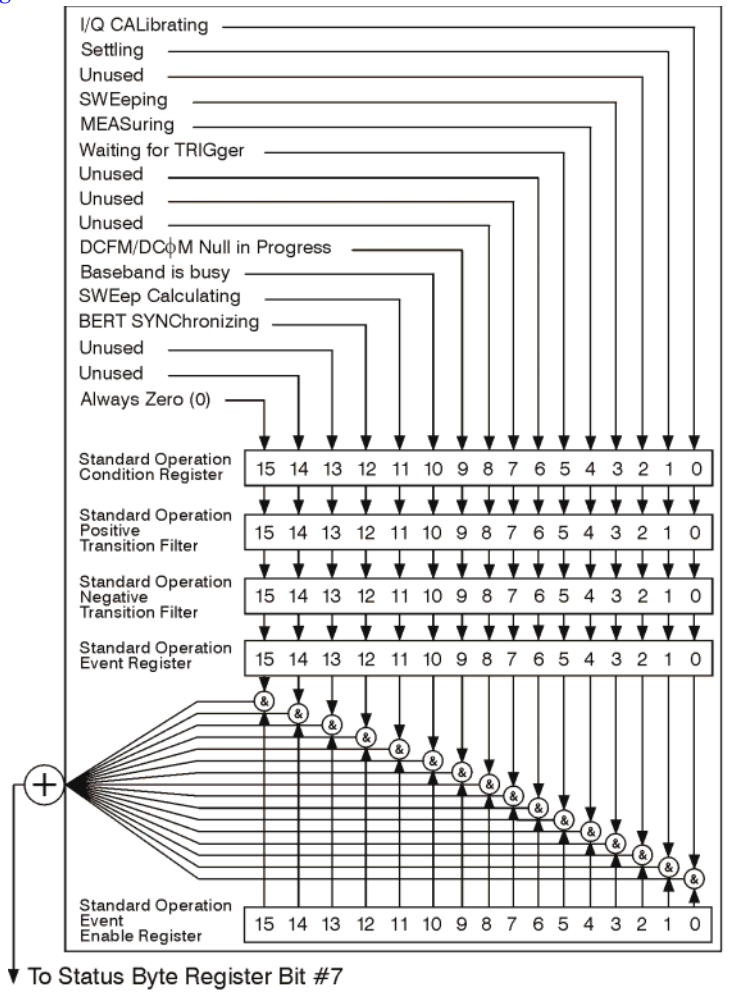
Standard Operation Status Group

NOTE Some of the bits in this status group do not apply to the E4428C, E8257D, E8267D, E8663B, and the N5161A/62A/81A/82A/83A, and returns zero when queried. See Table 4-6 on page 173 for more information.

The Agilent MXG SCPI command `:STAT:OPER:SUPP`, can suppress the managing of this status group and save 50 us from the switching time. Refer to the *SCPI Command Reference*.

The Operation Status Group is used to determine the specific event that set bit 7 in the Status Byte Register. This group consists of the Standard Operation Condition Register, the Standard Operation Transition Filters (negative and positive), the Standard Operation Event Register, and the Standard Operation Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Standard Operation Condition Register

The Standard Operation Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-6 Standard Operation Condition Register Bits

Bit	Description
0[a]	**I/Q Calibrating**. A 1 in this position indicates an I/Q calibration is in process.
1	**Settling**. A 1 in this bit position indicates that the signal generator is settling.
2	**Unused**. This bit position is always set to 0.
3	**Sweeping**. A 1 in this bit position indicates that a sweep is in progress.
4[b]	**Measuring**. A 1 in this bit position indicates that a bit error rate test is in progress.
5[c]	**Waiting for Trigger**. A 1 in this bit position indicates that the source is in a "wait for trigger" state. When option 300 is enabled, a 1 in this bit position indicates that TCH/PDCH synchronization is established and waiting for a trigger to start measurements.
6,7,8	**Unused**. These bits are always set to 0.
9[d]	**DCFM/DCϕM Null in Progress**. A 1 in this bit position indicates that the signal generator is currently performing a DCFM/DCΦM zero calibration.
10[c]	**Baseband is Busy**. A 1 in this bit position indicates that the baseband generator is communicating or processing. This is a summary bit. See the "Baseband Operation Status Group" on page 175 for more information.
11[e]	**Sweep Calculating**. A 1 in this bit position indicates that the signal generator is currently doing the necessary pre-sweep calculations.
12[b]	**BERT Synchronizing.** A 1 in this bit position is set while the BERT is synchronizing to 'BCH', then 'TCH' and then to 'PRBS'.
13, 14	**Unused**. These bits are always set to 0.
15	**Always 0**.

a. In the N5161A/81A, E4428C, E8257D, and E8663B, this bit is always set to 0.
b. Always set to 0 if Option UN7 not present (E4438C only).
c. Option 300 is only available on the E4438C.
d. This bit is always set to 0.
e. In the N5161A/81A and N5162A/82A this bit is always set to 0.

Query: `STATus:OPERation:CONDition?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 520 is returned. The decimal sum = 512 (bit 9) + 8 (bit 3).

Standard Operation Transition Filters (negative and positive)

The Standard Operation Transition Filters specify which types of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:OPERation:NTRansition <value>` (negative transition), or
`STATus:OPERation:PTRansition <value>` (positive transition), where
`<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:OPERation:NTRansition?`
`STATus:OPERation:PTRansition?`

Standard Operation Event Register

The Standard Operation Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read only. Reading data from an event register clears the content of that register.

Query: `STATus:OPERation[:EVENt]?`

Standard Operation Event Enable Register

The Standard Operation Event Enable Register lets you choose which bits in the Standard Operation Event Register set the summary bit (bit 7 of the Status Byte Register) to 1.

Command: `STATus:OPERation:ENABle <value>`, where
`<value>` is the sum of the decimal values of the bits you want to enable.

Example: To enable bit 9 and bit 3 so that whenever either of those bits are set to 1, the Standard Operation Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:OPER:ENAB 520` (512 + 8).

Query: `STATus:OPERation:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:OPERation:ENABle <value>` command.

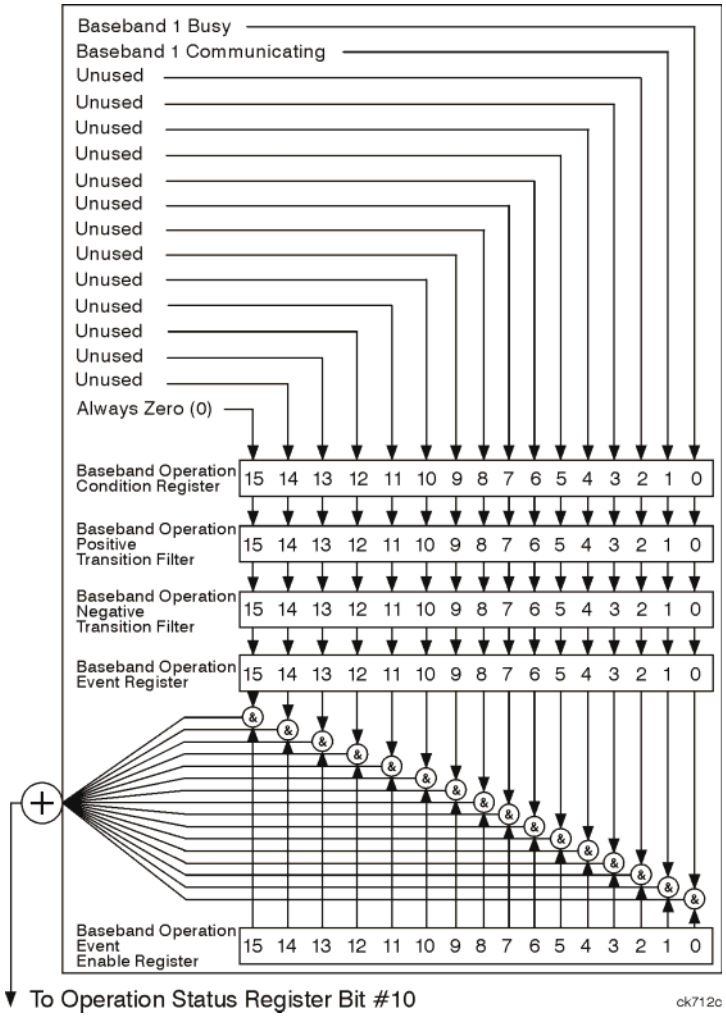
Baseband Operation Status Group

NOTE This status group does not apply to the E4428C, E8257D, and the E8663B, and if queried, returns zero. See Table 4-7 on page 176 for more information.

This status group does not apply to the N5161A/62A/81A/82A/83A. (If queried, the signal generator will not respond.)

The Baseband Operation Status Group is used to determine the specific event that set bit 10 in the Standard Operation Status Group. This group consists of the Baseband Operation Condition Register, the Baseband Operation Transition Filters (negative and positive), the Baseband Operation Event Register, and the Baseband Operation Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Baseband Operation Condition Register

The Baseband Operation Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-7 Baseband Operation Condition Register Bits

Bit	Description
0	**Baseband 1 Busy**. A 1 in this position indicates the signal generator baseband is active.
1	**Baseband 1 Communicating.** A 1 in this bit position indicates that the signal generator baseband generator is handling data IO.
2–14	**Unused**. This bit position is always set to 0.
15	**Always 0**.

Query: `STATus:OPERation:BASeband:CONDition?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 2 is returned. The decimal sum = 2 (bit 1).

Baseband Operation Transition Filters (negative and positive)

The Baseband Operation Transition Filters specify which types of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:OPERation:BASeband:NTRansition <value>` (negative transition), or
`STATus:OPERation:BASeband:PTRansition <value>` (positive transition), where
`<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:OPERation:BASeband:NTRansition?`
`STATus:OPERation:BASeband:PTRansition?`

Baseband Operation Event Register

The Baseband Operation Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read only. Reading data from an event register clears the content of that register.

Query: `STATus:OPERation:BASeband[:EVENt]?`

Baseband Operation Event Enable Register

The Baseband Operation Event Enable Register lets you choose which bits in the Baseband Operation Event Register can set the summary bit (bit 7 of the Status Byte Register).

Command: `STATus:OPERation:BASeband:ENABle <value>`, where `<value>` is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 0 and bit 1 so that whenever either of those bits are set to 1, the Baseband Operation Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:OPER:ENAB (2 + 1)`.

Query: `STATus:OPERation:BASeband:ENABle?`

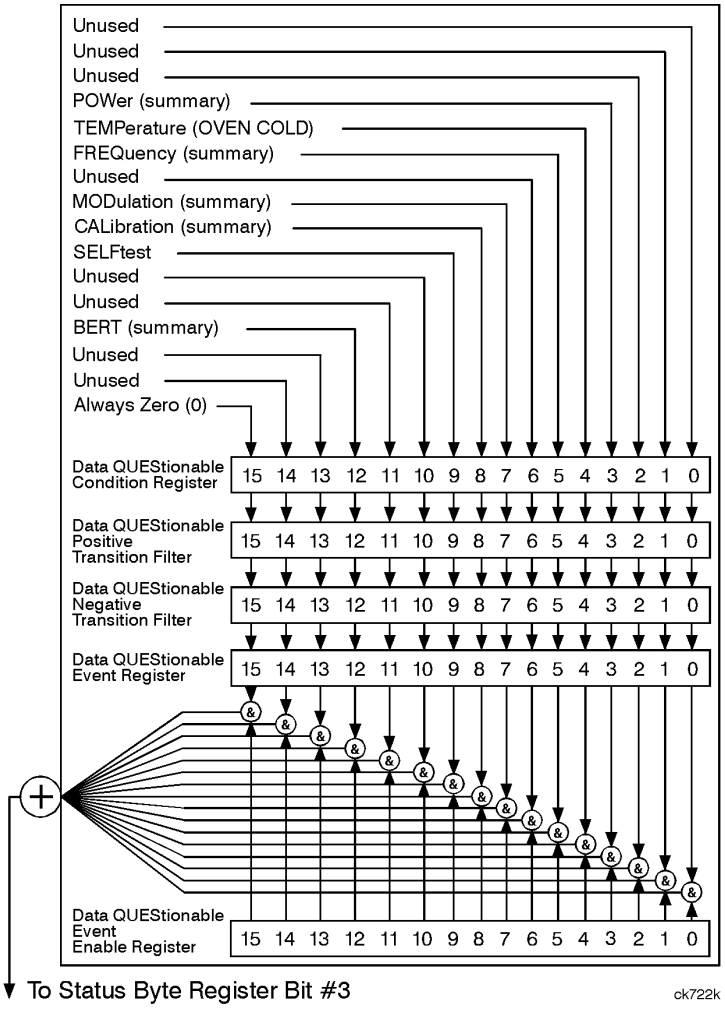
Response: Decimal value of the sum of the bits previously enabled with the `STATus:OPERation:BASeband:ENABle <value>` command.

Data Questionable Status Group

NOTE Some of the bits in this status group do not apply to the E4428C, E8257D, E8267D, E8663B, and the N5161A/62A/81A/82A/83A, and returns zero when queried. Other bits have changed state content. See Table 4-8 on page 179 for more information.

The Data Questionable Status Group is used to determine the specific event that set bit 3 in the Status Byte Register. This group consists of the Data Questionable Condition Register, the Data Questionable Transition Filters (negative and positive), the Data Questionable Event Register, and the Data Questionable Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Data Questionable Condition Register

The Data Questionable Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-8 Data Questionable Condition Register Bits

Bit	Description
0, 1, 2	**Unused**. These bits are always set to 0.
3	**Power (summary)**. This is a summary bit taken from the QUEStionable:POWer register. A 1 in this bit position indicates that one of the following may have happened: The ALC (Automatic Leveling Control) is unable to maintain a leveled RF output power (i.e., ALC is UNLEVELED), the reverse power protection circuit has been tripped. See the "Data Questionable Power Status Group" on page 181 for more information.
4	***N5161A/62A/81A/82A/83A*: ALC Heater Detector (COLD)**. A 1 in this bit position indicates that the ALC detector is cold. ***E4428C/38C, E8257D/67D*, and E8663B: Temperature (OVEN COLD)**. A 1 in this bit position indicates that the internal reference oscillator (reference oven) is cold.
5	**Frequency (summary)**. This is a summary bit taken from the QUEStionable:FREQuency register. A 1 in this bit position indicates that one of the following may have happened: synthesizer PLL unlocked, 10 MHz reference VCO PLL unlocked, 1 GHz reference unlocked, sampler, YO loop unlocked or baseband 1 unlocked. For more information, see the "Data Questionable Frequency Status Group" on page 184.
6	**Unused**. This bit is always set to 0.
7	**Modulation (summary)**. This is a summary bit taken from the QUEStionable:MODulation register. A 1 in this bit position indicates that one of the following may have happened: modulation source 1 underrange, modulation source 1 overrange, modulation source 2 underrange, modulation source 2 overrange, or modulation uncalibrated. See the "Data Questionable Modulation Status Group" on page 187 for more information.
8[ab]	**Calibration (summary)**. This is a summary bit taken from the QUEStionable:CALibration register. A 1 in this bit position indicates that one of the following may have happened: an error has occurred in the DCFM/DCΦM zero calibration, or an error has occurred in the I/Q calibration. See the "Data Questionable Calibration Status Group" on page 190 for more information.
9	**Self Test**. A 1 in this bit position indicates that a self-test has failed during power-up. Reset this bit by cycling the signal generator's line power. *CLS will not clear this bit.
10, 11	**Unused**. These bits are always set to 0.
12[c]	**BERT (summary).** This is a summary bit taken from the QUEStionable:BERT register. A 1 in this bit position indicates that one of the following occurred: no BCH/TCH synchronization, no data change, no clock input, PRBS not synchronized, demod/DSP unlocked, or demod unleveled. See the "Data Questionable BERT Status Group" on page 193 for more information.
13, 14	**Unused**. These bits are always set to 0.
15	**Always 0**.

a. In the N5162A/82A, this bit applies only to the I/Q calibration. In the N5161A/81A/83A, this bit is unused and always set to 0.
b. In the E8257D, and the E8663B this bit applies only to the DCFM/DCΦM calibration.
c. In the N5161A/81A, N5162A/82A, N5183A, E4428C, E8257D, E8267D, and the E8663B this bit is always set to 0.

Query: `STATus:QUEStionable:CONDition?`

Response: The *decimal* sum of the bits set to 1

Example: The decimal value 520 is returned. The decimal sum = 512 (bit 9) + 8 (bit 3).

Data Questionable Transition Filters (negative and positive)

The Data Questionable Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:NTRansition <value>` (negative transition), or `STATus:QUEStionable:PTRansition <value>` (positive transition), where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries:
```
STATus:QUEStionable:NTRansition?
STATus:QUEStionable:PTRansition?
```

Data Questionable Event Register

The Data Questionable Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable[:EVENt]?`

Data Questionable Event Enable Register

The Data Questionable Event Enable Register lets you choose which bits in the Data Questionable Event Register set the summary bit (bit 3 of the Status Byte Register) to 1.

Command: `STATus:QUEStionable:ENABle <value>` where `<value>` is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 9 and bit 3 so that whenever either of those bits are set to 1, the Data Questionable Status summary bit of the Status Byte Register is set to 1. Send the command `STAT:QUES:ENAB 520` (512 + 8).

Query: `STATus:QUEStionable:ENABle?`

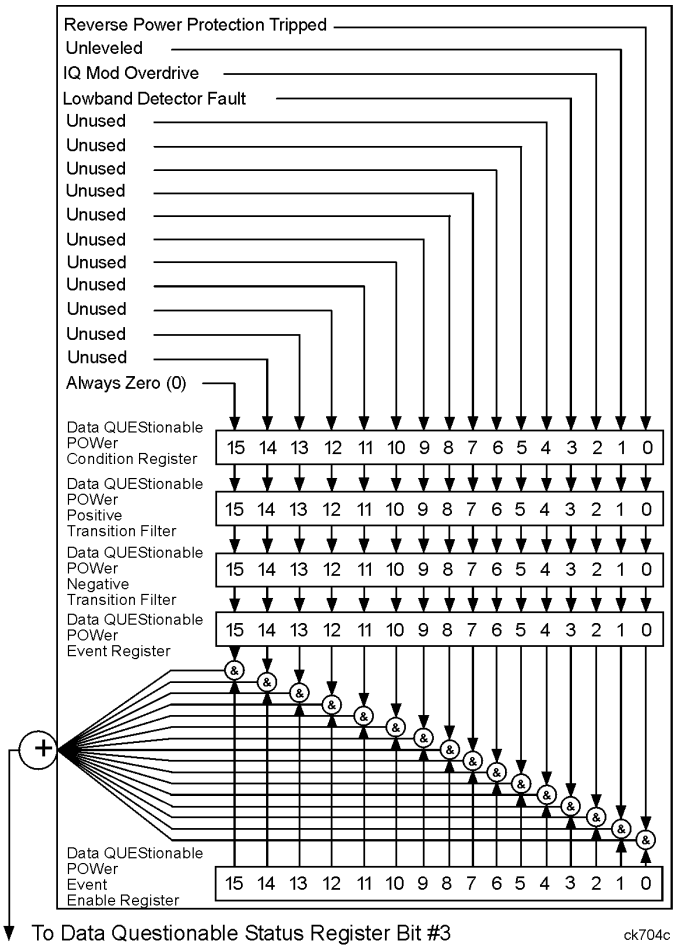
Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:ENABle <value>` command.

Data Questionable Power Status Group

NOTE Some of the bits in this status group do not apply to the E4428C, E8257D, E8267D, E8663B, and the N5161A/62A/81A/82A/83A, and returns zero when queried. See Table 4-9 on page 182 for more information.

The Data Questionable Power Status Group is used to determine the specific event that set bit 3 in the Data Questionable Condition Register. This group consists of the Data Questionable Power Condition Register, the Data Questionable Power Transition Filters (negative and positive), the Data Questionable Power Event Register, and the Data Questionable Power Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Data Questionable Power Condition Register

The Data Questionable Power Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-9 Data Questionable Power Condition Register Bits

Bit	Description
0[a]	**Reverse Power Protection Tripped**. A 1 in this bit position indicates that the reverse power protection (RPP) circuit has been tripped. There is no output in this state. Any conditions that may have caused the problem should be corrected. Reset the RPP circuit by sending the remote SCPI command: OUTput:PROTection:CLEar. Resetting the RPP circuit bit, resets this bit to 0.
1	**Unleveled**. A 1 in this bit position indicates that the output leveling loop is unable to set the output power.
2[b]	**IQ Mod Overdrive**. A 1 in this bit position indicates that the signal level into the IQ modulator is too high.
3[c]	**Lowband Detector Fault**. A 1 in this bit position indicates that the lowband detector heater circuit has failed.
4–14	**Unused**. These bits are always set to 0.
15	**Always 0.**

a. In the N5161A/62A/81A/82A with Option 506, the N5183A, and the E4428C/38C with Option 506, this bit is set to 0.
b. In the N5161A/62A/81A/82A/83A, E4428C, E8257D/67D, and E8663B, this bit is set to 0.

Query: `STATus:QUEStionable:POWer:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable Power Transition Filters (negative and positive)

The Data Questionable Power Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:POWer:NTRansition <value>` (negative transition), or `STATus:QUEStionable:POWer:PTRansition <value>` (positive transition), where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:POWer:NTRansition?` `STATus:QUEStionable:POWer:PTRansition?`

Data Questionable Power Event Register

The Data Questionable Power Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:POWer[:EVENt]?`

Data Questionable Power Event Enable Register

The Data Questionable Power Event Enable Register lets you choose which bits in the Data Questionable Power Event Register set the summary bit (bit 3 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:POWer:ENABle <value>` where `<value>` is the sum of the decimal values of the bits you want to enable

Example: Enable bit 3 and bit 2 so that whenever either of those bits are set to 1, the Data Questionable Power summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:POW:ENAB 520` (8 + 4).

Query: `STATus:QUEStionable:POWer:ENABle?`

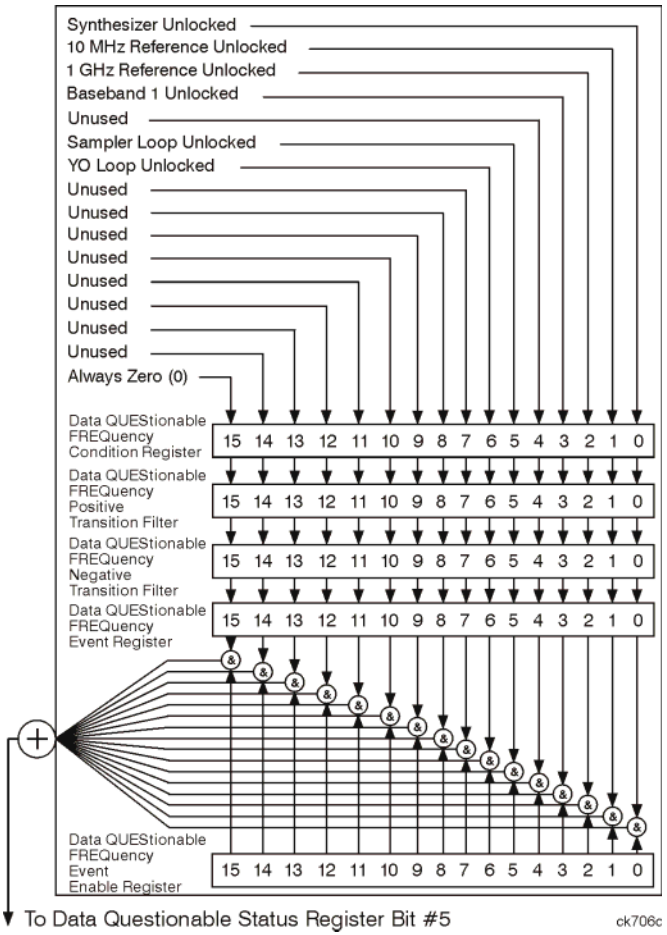
Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:POWer:ENABle <value>` command.

Data Questionable Frequency Status Group

NOTE Some bits in this status group do not apply to the N5161A/62A/81A/82A/83A, E4428C, E8257D, and the E8663B and returns zero when queried. See Table 4-10 on page 185 for more information.

The Data Questionable Frequency Status Group is used to determine the specific event that set bit 5 in the Data Questionable Condition Register. This group consists of the Data Questionable Frequency Condition Register, the Data Questionable Frequency Transition Filters (negative and positive), the Data Questionable Frequency Event Register, and the Data Questionable Frequency Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Data Questionable Frequency Condition Register

The Data Questionable Frequency Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read-only.

Table 4-10 Data Questionable Frequency Condition Register Bits

Bit	Description
0	**Synth. Unlocked**. A 1 in this bit position indicates that the synthesizer is unlocked.
1	**10 MHz Ref Unlocked**. A 1 in this bit position indicates that the 10 MHz reference signal is unlocked.
2[a]	**1 GHz Ref Unlocked**. A 1 in this bit position indicates that the 1 GHz reference signal is unlocked.
3[b]	**Baseband 1 Unlocked**. A 1 in this bit position indicates that the baseband generator is unlocked.
4	**Unused**. This bit is always set to 0.
5[b]	**Sampler Loop Unlocked**. A 1 in this bit position indicates that the sampler loop is unlocked.
6[b]	**YO Loop Unlocked**. A 1 in this bit position indicates that the YO loop is unlocked.
7–14	**Unused**. These bits are always set to 0.
15	**Always 0**.

a. In the N5161A/81A and N5162A/82A these bits are always set to 0.
b. In the N5161A/62A/81A/82A/83A, E4428C, E8257D, and the E8663B, this bit is always set to 0.

Query: `STATus:QUEStionable:FREQuency:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable Frequency Transition Filters (negative and positive)

Specifies which types of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:FREQuency:NTRansition <value>` (negative transition) or
`STATus:QUEStionable:FREQuency:PTRansition <value>` (positive transition) where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:FREQuency:NTRansition?`
`STATus:QUEStionable:FREQuency:PTRansition?`

Data Questionable Frequency Event Register

Latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:FREQuency[:EVENt]?`

Data Questionable Frequency Event Enable Register

Lets you choose which bits in the Data Questionable Frequency Event Register set the summary bit (bit 5 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:FREQuency:ENABle <value>`, where `<value>` is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 4 and bit 3 so that whenever either of those bits are set to 1, the Data Questionable Frequency summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:FREQ:ENAB 520` (16 + 8).

Query: `STATus:QUEStionable:FREQuency:ENABle?`

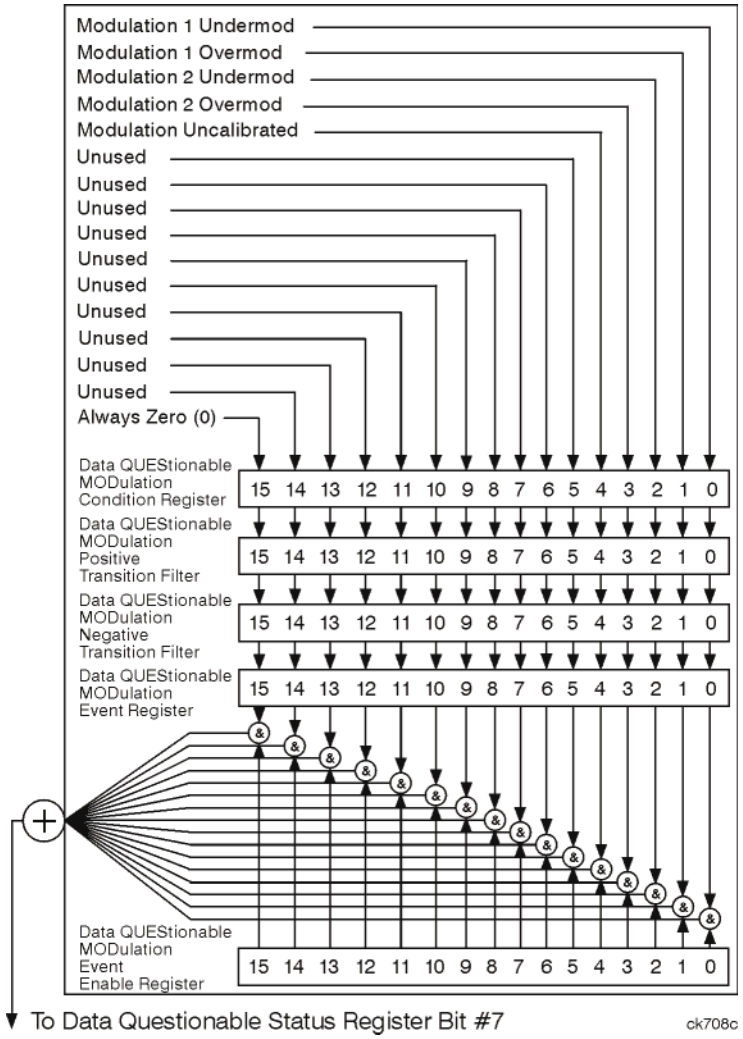
Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:FREQuency:ENABle <value>` command.

Data Questionable Modulation Status Group

NOTE This status group does not apply to the N5161A/81A and the N5162A/82A, and returns zero when queried. See Table 4-11 on page 188 for more information.

The Data Questionable Modulation Status Group is used to determine the specific event that set bit 7 in the Data Questionable Condition Register. This group consists of the Data Questionable Modulation Condition Register, the Data Questionable Modulation Transition Filters (negative and positive), the Data Questionable Modulation Event Register, and the Data Questionable Modulation Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Data Questionable Modulation Condition Register

The Data Questionable Modulation Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read-only.

Table 4-11 Data Questionable Modulation Condition Register Bits

Bit	Description
0	**Modulation 1 Undermod**. A 1 in this bit position indicates that the External 1 input, ac coupling on, is less than 0.97 volts.
1	**Modulation 1 Overmod**. A 1 in this bit position indicates that the External 1 input, ac coupling on, is more than 1.03 volts.
2	**Modulation 2 Undermod**. A 1 in this bit position indicates that the External 2 input, ac coupling on, is less than 0.97 volts.
3	**Modulation 2 Overmod**. A 1 in this bit position indicates that the External 2 input, ac coupling on, is more than 1.03 volts.
4	**Modulation Uncalibrated**. A 1 in this bit position indicates that modulation is uncalibrated.
5–14	**Unused**. This bit is always set to 0.
15	**Always 0**.

Query: `STATus:QUEStionable:MODulation:CONDition?`

Response: The *decimal* sum of the bits set to 1

Data Questionable Modulation Transition Filters (negative and positive)

The Data Questionable Modulation Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:MODulation:NTRansition <value>` (negative transition), or
`STATus:QUEStionable:MODulation:PTRansition <value>` (positive transition), where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:MODulation:NTRansition?`
`STATus:QUEStionable:MODulation:PTRansition?`

Data Questionable Modulation Event Register

The Data Questionable Modulation Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:MODulation[:EVENt]?`

Data Questionable Modulation Event Enable Register

The Data Questionable Modulation Event Enable Register lets you choose which bits in the Data Questionable Modulation Event Register set the summary bit (bit 7 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:MODulation:ENABle <value>` where `<value>` is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 4 and bit 3 so that whenever either of those bits are set to 1, the Data Questionable Modulation summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:MOD:ENAB 520` (16 + 8).

Query: `STATus:QUEStionable:MODulation:ENABle?`

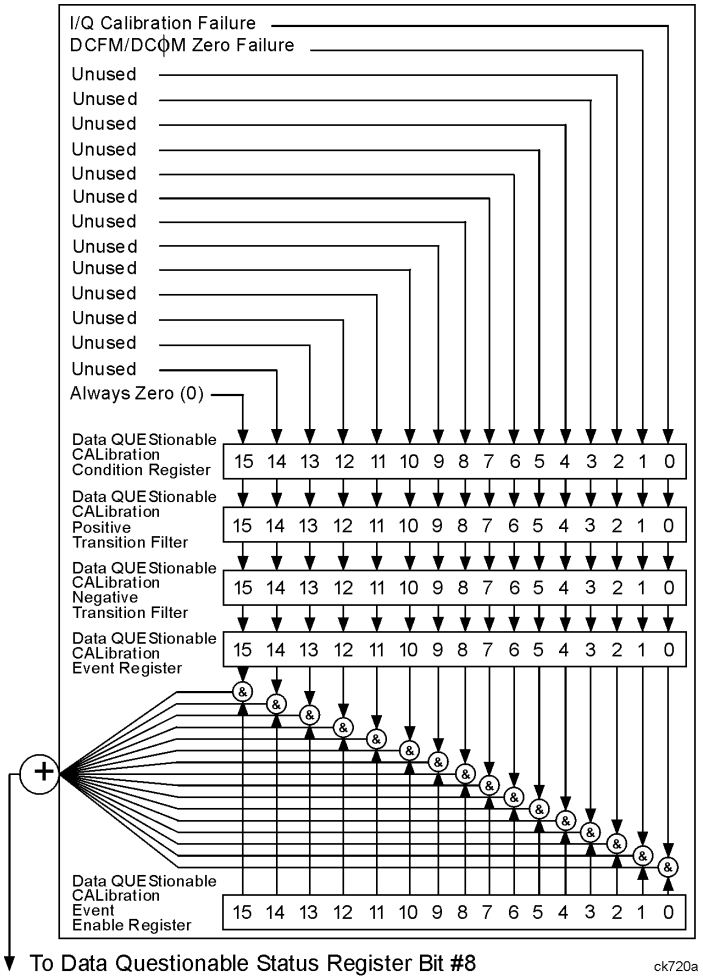
Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:MODulation:ENABle <value>` command.

Data Questionable Calibration Status Group

NOTE Some bits in this status group do not apply to the N5161A/62A/81A/82A/83A, E4428C, E8257D, and the E8663B, and return zero when queried. See Table 4-12 on page 191 for more information.

The Data Questionable Calibration Status Group is used to determine the specific event that set bit 8 in the Data Questionable Condition Register. This group consists of the Data Questionable Calibration Condition Register, the Data Questionable Calibration Transition Filters (negative and positive), the Data Questionable Calibration Event Register, and the Data Questionable Calibration Event Enable Register.

This is the named status register for the E4438C. However, not all signal generator models use all of the shown events (i.e. some use only a subset of the E4438C's status registers).



Data Questionable Calibration Condition Register

The Data Questionable Calibration Condition Register continuously monitors the calibration status of the signal generator. Condition registers are read only.

Table 4-12 Data Questionable Calibration Condition Register Bits

Bit	Description
0[a]	**I/Q Calibration Failure**. A 1 in this bit position indicates that the I/Q modulation calibration experienced a failure.
1[b]	**DCFM/DCΦM Zero Failure**. A 1 in this bit position indicates that the DCFM/DCΦM zero calibration routine has failed. This is a critical error. The output of the source has no validity until the condition of this bit is 0.
2–14	**Unused**. These bits are always set to 0.
15	**Always 0**.

a. In the N5161A/81A, E4428C, E8257D, and the E8663B, this bit is set to 0.
b. In the N5161A/81A and N5162A/82A, this bit is set to 0.

Query: `STATus:QUEStionable:CALibration:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable Calibration Transition Filters (negative and positive)

The Data Questionable Calibration Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:CALibration:NTRansition <value>` (negative transition), or
`STATus:QUEStionable:CALibration:PTRansition <value>` (positive transition), where `<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:CALibration:NTRansition?`
`STATus:QUEStionable:CALibration:PTRansition?`

Data Questionable Calibration Event Register

The Data Questionable Calibration Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:CALibration[:EVENt]?`

Data Questionable Calibration Event Enable Register

The Data Questionable Calibration Event Enable Register lets you choose which bits in the Data Questionable Calibration Event Register set the summary bit (bit 8 of the Data Questionable Condition register) to 1.

Command: `STATus:QUEStionable:CALibration:ENABle <value>`, where `<value>` is the sum of the decimal values of the bits you want to enable.

Example: Enable bit 1 and bit 0 so that whenever either of those bits are set to 1, the Data Questionable Calibration summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:CAL:ENAB 520` (2 + 1).

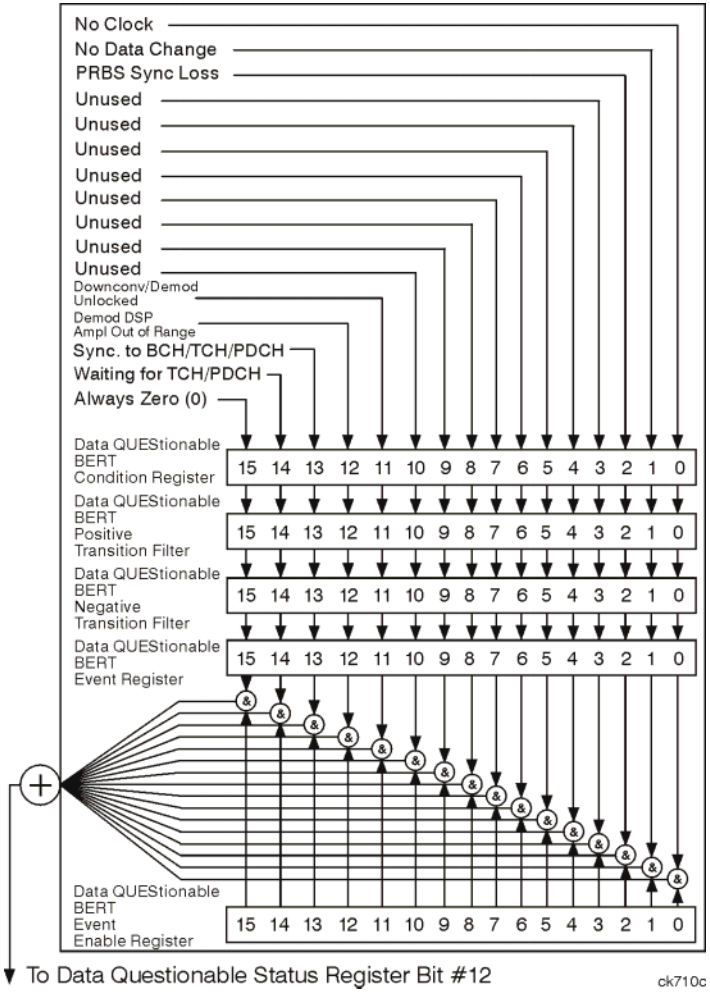
Query: `STATus:QUEStionable:CALibration:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:CALibration:ENABle <value>` command.

Data Questionable BERT Status Group

NOTE This status group applies only to the E4438C with Option UN7. Refer to Table 4-13 on page 194 for more information.

The Data Questionable BERT Status Group is used to determine the specific event that set bit 12 in the Data Questionable Condition Register. The Data Questionable Status group consists of the Data Questionable BERT Condition Register, the Data Questionable BERT Transition Filters (negative and positive), the Data Questionable BERT Event Register, and the Data Questionable BERT Event Enable Register.



Data Questionable BERT Condition Register

The Data Questionable BERT Condition Register continuously monitors the hardware and firmware status of the signal generator. Condition registers are read only.

Table 4-13 Data Questionable BERT Condition Register Bits

Bit	Description
0	**No Clock**. A 1 in this bit position indicates no clock input for more than 3 seconds.
1	**No Data Change**. A 1 in this bit position indicates no data change occurred during the last 200 clock signals.
2	**PRBS Sync Loss**. A 1 is set while PRBS synchronization is not established. `*RST` sets the bit to zero.
3–10	**Unused**. These bits are always set to 0.
11	**Down conv. / Demod Unlocked**. A 1 in this bit position indicates that either the demodulator or the down converter is out of lock.
12	**Demod DSP Ampl out of range**. A 1 in this bit position indicates the demodulator amplitude is out of range. The ***RST** command sets this bit to zero (0).
13	**Sync. to BCH/TCH/PDCH**. If the synchronization source is BCH, a 1 in this bit position indicates BCH synchronization is not established; it does not indicate the TCH/PDCH synchronization status. If the sync source is TCH or PDCH, a 1 in this bit position indicates that TCH or PDCH synchronization is not established. ***RST** sets this bit to zero.
14	**Waiting for TCH/PDCH**. A 1 in this bit position indicates that a TCH or PDCH midamble has not been received. This bit is set when bit 13 is set. The bit is also set when the TCH or PDCH synchronization was once locked and then lost (in this case the front panel displays "WAITING FOR TCH (or PDCH)"). ***RST** sets this bit to zero.
15	**Always 0.**

Query: `STATus:QUEStionable:BERT:CONDition?`

Response: The *decimal* sum of the bits set to 1.

Data Questionable BERT Transition Filters (negative and positive)

The Data Questionable BERT Transition Filters specify which type of bit state changes in the condition register set corresponding bits in the event register. Changes can be positive (0 to 1) or negative (1 to 0).

Commands: `STATus:QUEStionable:BERT:NTRansition <value>` (negative transition), or
`STATus:QUEStionable:BERT:PTRansition <value>` (positive transition), where
`<value>` is the sum of the decimal values of the bits you want to enable.

Queries: `STATus:QUEStionable:BERT:NTRansition? STATus:QUEStionable:BERT:PTRansition?`

Data Questionable BERT Event Register

The Data Questionable BERT Event Register latches transition events from the condition register as specified by the transition filters. Event registers are destructive read-only. Reading data from an event register clears the content of that register.

Query: `STATus:QUEStionable:BERT[:EVENt]?`

Data Questionable BERT Event Enable Register

The Data Questionable BERT Event Enable Register lets you choose which bits in the Data Questionable BERT Event Register set the summary bit (bit 3 of the Data Questionable Condition Register) to 1.

Command: `STATus:QUEStionable:BERT:ENABle <value>` where `<value>` is the sum of the decimal values of the bits you want to enable

Example: Enable bit 11 and bit 2 so that whenever either of those bits are set to 1, the Data Questionable BERT summary bit of the Data Questionable Condition Register is set to 1. Send the command `STAT:QUES:BERT:ENAB 520` (2048 + 4).

Query: `STATus:QUEStionable:BERT:ENABle?`

Response: Decimal value of the sum of the bits previously enabled with the `STATus:QUEStionable:BERT:ENABle <value>` command.

5 Creating and Downloading Waveform Files

NOTE The ability to play externally created waveform data in the signal generator is available only in the N5162A/82A with Option 651, 652 or 654, E4438C ESG Vector Signal Generators with Option 001, 002, 601, or 602, and E8267D PSG Vector Signal Generators with Option 601 or 602.

On the Agilent MXG, the internal baseband generator speed upgrade Options 670, 671, and 672 are option upgrades that *require* Option 651 and 652 to have been loaded at the factory (refer to the *Data Sheet* for more information). Any references to 651, 652, or 654 are inclusive of 671, 672, and 674.

For the N5161A/62A the softkey menus and features mentioned in this chapter are only available through the Web–Enabled MXG or via SCPI commands. Refer to "Using the Web Browser" on page 11 and to the *SCPI Command Reference*.

This chapter explains how to create Arb–based waveform data and download it into the signal generator.

- "Overview of Downloading and Extracting Waveform Files" on page 198
- "Understanding Waveform Data" on page 199
- "Waveform Structure" on page 207
- "Waveform Phase Continuity" on page 209
- "Waveform Memory" on page 212
- "Commands for Downloading and Extracting Waveform Data" on page 218
- "Creating Waveform Data" on page 227
- "Downloading Waveform Data" on page 233
- "Loading, Playing, and Verifying a Downloaded Waveform" on page 239
- "Using the Download Utilities" on page 242
- "Downloading E443xB Signal Generator Files" on page 243
- "Programming Examples" on page 246
- "Troubleshooting Waveform Files" on page 287

Overview of Downloading and Extracting Waveform Files

The signal generator lets you download and extract waveform files. You can create these files either external to the signal generator or by using one of the internal modulation formats (ESG/PSG only). The signal generator also accepts waveforms files created for the earlier E443xB ESG signal generator models. For file extractions, the signal generator encrypts the waveform file information. The exception to encrypted file extraction is user–created I/Q data. The signal generator lets you extract this type of file unencrypted. After extracting a waveform file, you can download it into another Agilent signal generator that has the same option or software license required to play it. Waveform files consist of three items:

1. I/Q data
2. Marker data
3. File header

NOTE This order of download is required, as the I/Q data downloads results in the overwriting of all of these three parts of the file.

The signal generator automatically creates the marker file and the file header if the two items are *not* part of the download. In this situation, the signal generator sets the file header information to unspecified (no settings saved) and sets all markers to zero (off).

There are three ways to download waveform files: FTP, programmatically or using one of three available free download utilities created by Agilent Technologies:

- N7622A Signal Studio Toolkit 2
  http://www.agilent.com/find/signalstudio
- Agilent Waveform Download Assistant for use only with MATLAB
  http://www.agilent.com/find/downloadassistant
- Intuilink for Agilent PSG/ESG Signal Generators
  http://www.agilent.com/find/intuilink

NOTE Agilent Intuilink is *not* available for the Agilent MXG.

Waveform Data Requirements

To be successful in downloading files, you must first create the data in the required format.

- Signed 2's complement
- 2–byte integer values
- Input data range of –32768 to 32767
- Minimum of 60 samples per waveform (60 I and 60 Q data points)
- Interleaved I and Q data
- Big endian byte order
- The same name for the marker, header, and I/Q file

  This is only a requirement if you create and download a marker file and or file header, otherwise the signal generator automatically creates the marker file and or file header using the I/Q data file name.

NOTE FTP can be used without programming commands to transfer files from the PC to the signal generator or from the signal generator to the PC.

For more information, see "Waveform Structure" on page 207.

For more information on waveform data, see "Understanding Waveform Data" on page 199.

Understanding Waveform Data

The signal generator accepts binary data formatted into a binary I/Q file. This section explains the necessary components of the binary data, which uses ones and zeros to represent a value.

Bits and Bytes

Binary data uses the base–two number system. The location of each bit within the data represents a value that uses base two raised to a power (2^{n-1}). The exponent is n – 1 because the first position is zero. The first bit position, zero, is located at the far right. To find the decimal value of the binary data, sum the value of each location:

$$1101 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$
$$= (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$$
$$= 13 \text{ (decimal value)}$$

Notice that the exponent identifies the bit position within the data, and we read the data from right to left.

The signal generator accepts data in the form of bytes. Bytes are groups of eight bits:

$$01101110 = (0 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$
$$= 110 \text{ (decimal value)}$$

The maximum value for a single unsigned byte is 255 (11111111 or 2^8–1), but you can use multiple bytes to represent larger values. The following shows two bytes and the resulting integer value:

01101110 10110011= 28339 (decimal value)

The maximum value for two unsigned bytes is 65535. Since binary strings lengthen as the value increases, it is common to show binary values using hexadecimal (hex) values (base 16), which are shorter. The value 65535 in hex is FFFF. Hexadecimal consists of the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In decimal, hex values range from 0 to 15 (F). It takes 4 bits to represent a single hex value.

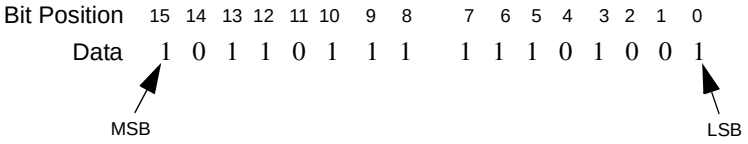
1 = 0001	2 = 0010	3 = 0011	4 = 0100	5 = 0101
6 = 0110	7 = 0111	8 = 1000	9 = 1001	A = 1010
B = 1011	C = 1100	D = 1101	E = 1110	F = 1111

For I and Q data, the signal generator uses two bytes to represent an integer value.

LSB and MSB (Bit Order)

Within groups (strings) of bits, we designate the order of the bits by identifying which bit has the highest value and which has the lowest value by its location in the bit string. The following is an example of this order.

Most Significant Bit (MSB)	This bit has the highest value (greatest weight) and is located at the far left of the bit string.
Least Significant Bit (LSB)	This bit has the lowest value (bit position zero) and is located at the far right of the bit string.



Because we are using 2 bytes of data, the LSB appears in the second byte.

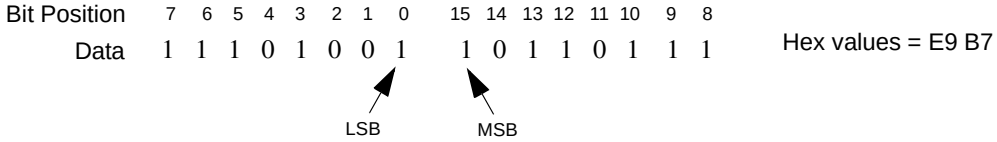
Little Endian and Big Endian (Byte Order)

When you use multiple bytes (as required for the waveform data), you must identify their order. This is similar to identifying the order of bits by LSB and MSB. To identify byte order, use the terms little endian and big endian. These terms are used by designers of computer processors.

Intel is a registered trademark of Intel Corporation.

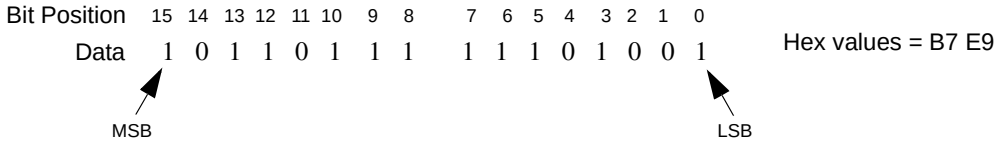
Little Endian Order

The lowest order byte that contains bits 0–7 comes first.



Big Endian Order

The highest order byte that contains bits 8–15 comes first.



Notice in the previous figure that the LSB and MSB positioning changes with the byte order. In little endian order, the LSB and MSB are next to each other in the bit sequence.

NOTE For I/Q data downloads, the signal generator requires big endian order. For each I/Q data point, the signal generator uses four bytes (two integer values), two bytes for the I point and two bytes for the Q point.

The byte order, little endian or big endian, depends on the type of processor used with your development platform. Intel processors and its clones use little endian. (Intel© is a U.S. registered trademark of Intel Corporation.) Sun™ and Motorola processors use big endian. The Apple PowerPC processor, while big endian oriented, also supports the little endian order. Always refer to the processor's manufacturer to determine the order they use for bytes and if they support both, to understand how to ensure that you are using the correct byte order.

Development platforms include any product that creates and saves waveform data to a file. This includes Agilent Technologies Advanced Design System EDA software, C++, MATLAB, and so forth.

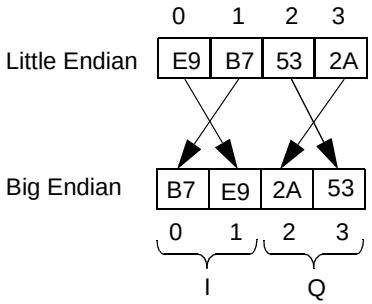
The byte order describes how the system processor stores integer values as binary data in memory. If you output data from a little endian system to a text file (ASCII text), the values are the same as viewed from a big endian system. The order only becomes important when you use the data in binary format, as is done when downloading data to the signal generator.

Sun is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

Byte Swapping

While the processor for the development platform determines the byte order, the recipient of the data may require the bytes in the reverse order. In this situation, you must reverse the byte order before downloading the data. This is commonly referred to as byte swapping. You can swap bytes either programmatically or by using either the Agilent Technologies Intuilink for ESG/PSG Signal Generator software, or the Signal Studio Toolkit 2 software. For the signal generator, byte swapping is the method to change the byte order of little endian to big endian. For more information on little endian and big endian order, see "Little Endian and Big Endian (Byte Order)" on page 200.

The following figure shows the concept of byte swapping for the signal generator. Remember that we can represent data in hex format (4 bits per hex value), so each byte (8 bits) in the figure shows two example hex values.



16–bit integer values (2 bytes = 1 integer value)

I data = bytes 0 and 1
Q data = bytes 2 and 3

To correctly swap bytes, you must group the data to maintain the I and Q values. One common method is to break the two–byte integer into one–byte character values (0–255). Character values use 8 bits (1 byte) to identify a character. Remember that the maximum unsigned 8–bit value is 255 ($2^8 - 1$). Changing the data into character codes groups the data into bytes. The next step is then to swap the bytes to align with big endian order.

NOTE The signal generator always assumes that downloaded data is in big endian order, so there is no data order check. Downloading data in little endian order will produce an undesired output signal.

DAC Input Values

The signal generator uses a 16–bit DAC (digital–to–analog convertor) to process each of the 2–byte integer values for the I and Q data points. The DAC determines the range of input values required from the I/Q data. Remember that with 16 bits we have a range of 0–65535, but the signal generator divides this range between positive and negative values:

- 32767 = positive full scale output
- 0 = 0 volts
- –32768 = negative full scale output

Because the DAC's range uses both positive and negative values, the signal generator requires signed input values. The following list illustrates the DAC's input value range.

Voltage	DAC Range	Input Range	Binary Data	Hex Data
Vmax	65535	32767	01111111 11111111	7FFF
	32768	1	00000000 00000001	0001
0 Volts	32767	0	00000000 00000000	0000
	32766	-1	11111111 11111111	FFFF
Vmin	0	–32768	10000000 00000000	8000

Notice that it takes only 15 bits (2^{15}) to reach the Vmax (positive) or Vmin (negative) values. The MSB determines the sign of the value. This is covered in "2's Complement Data Format" on page 205.

Using E443xB ESG DAC Input Values

In this section, the words *signal generator* with or without a model number refer to an N5162A/82A Agilent MXG, E4438C ESG, E8267D PSG. The signal generator input values differ from those of the earlier E443xB ESG models. For the E443xB models, the input values are all positive (unsigned) and the data is contained within 14 bits plus 2 bits for markers. This means that the E443xB DAC has a smaller range:

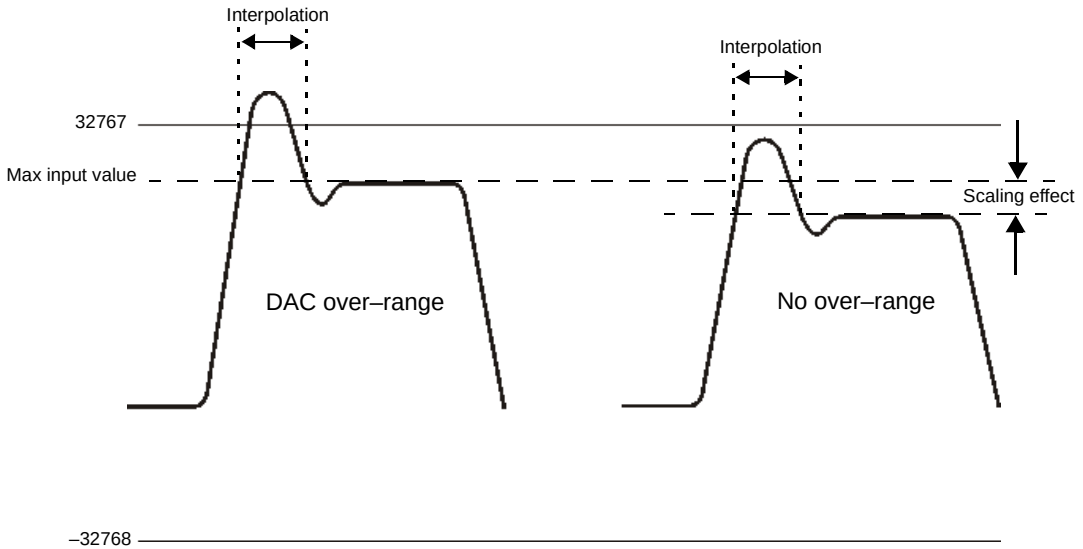
- 0 = negative full scale output
- 8192 = 0 volts
- 16383 = positive full scale output

Although the signal generator uses signed input values, it accepts unsigned data created for the E443xB and converts it to the proper DAC values. To download an E443xB files to the signal generator, use the same command syntax as for the E443xB models. For more information on downloading E443xB files, see "Downloading E443xB Signal Generator Files" on page 243.

Scaling DAC Values

The signal generator uses an interpolation algorithm (sampling between the I/Q data points) when reconstructing the waveform. For common waveforms, this interpolation can cause overshoot, which may exceed the limits of the signal process path's internal number representation, causing arithmatic overload. This will be reported as either a data path overload error (N5162A/82A) or a DAC over–range error condition (E4438C/E8267D). Because of the interpolation, the error condition can occur even when all the I and Q values are within the DAC input range. To avoid the DAC over–range problem, you must scale (reduce) the I and Q input values, so that any overshoot remains within the DAC range.

NOTE Whenever you interchange files between signal generator models, ensure that all scaling is adequate for that signal generator's waveform.



There is no single scaling value that is optimal for all waveforms. To achieve the maximum dynamic range, select the largest scaling value that does not result in a DAC over–range error. There are two ways to scale the I/Q data:

- Reduce the input values for the DAC.
- Use the SCPI command `:RADio:ARB:RSCaling <val>` to set the waveform amplitude as a percentage of full scale.

NOTE The signal generator factory preset for scaling is 70%. If you reduce the DAC input values, ensure that you set the signal generator scaling (:RADio:ARB:RSCaling) to an appropriate setting that accounts for the reduced values.

To further minimize overshoot problems, use the correct FIR filter for your signal type and adjust your sample rate to accommodate the filter response.

NOTE FIR filter capability is only available on the N5162A/82A with Option 651, 652, or 654, the E4438C with Option 001, 002, 601, or 602, and on the E8267D with Option 601 or 602.

2's Complement Data Format

The signal generator requires signed values for the input data. For binary data, two's complement is a way to represent positive and negative values. The most significant bit (MSB) determines the sign.

- 0 equals a positive value (01011011 = 91 decimal)
- 1 equals a negative value (10100101 = –91 decimal)

Like decimal values, if you sum the binary positive and negative values, you get zero. The one difference with binary values is that you have a carry, which is ignored. The following shows how to calculate the two's complement using 16–bits. The process is the same for both positive and negative values.

Convert the decimal value to binary.

```
23710 = 01011100 10011110
```

Notice that 15 bits (0–14) determine the value and bit 16 (MSB) indicates a positive value.
Invert the bits (1 becomes 0 and 0 becomes 1).

```
10100011 01100001
```

Add one to the inverted bits. Adding one makes it a two's complement of the original binary value.

```
  10100011 01100001
+ 00000000 00000001
  10100011 01100010
```

The MSB of the resultant is one, indicating a negative value (–23710).
Test the results by summing the binary positive and negative values; when correct, they produce zero.

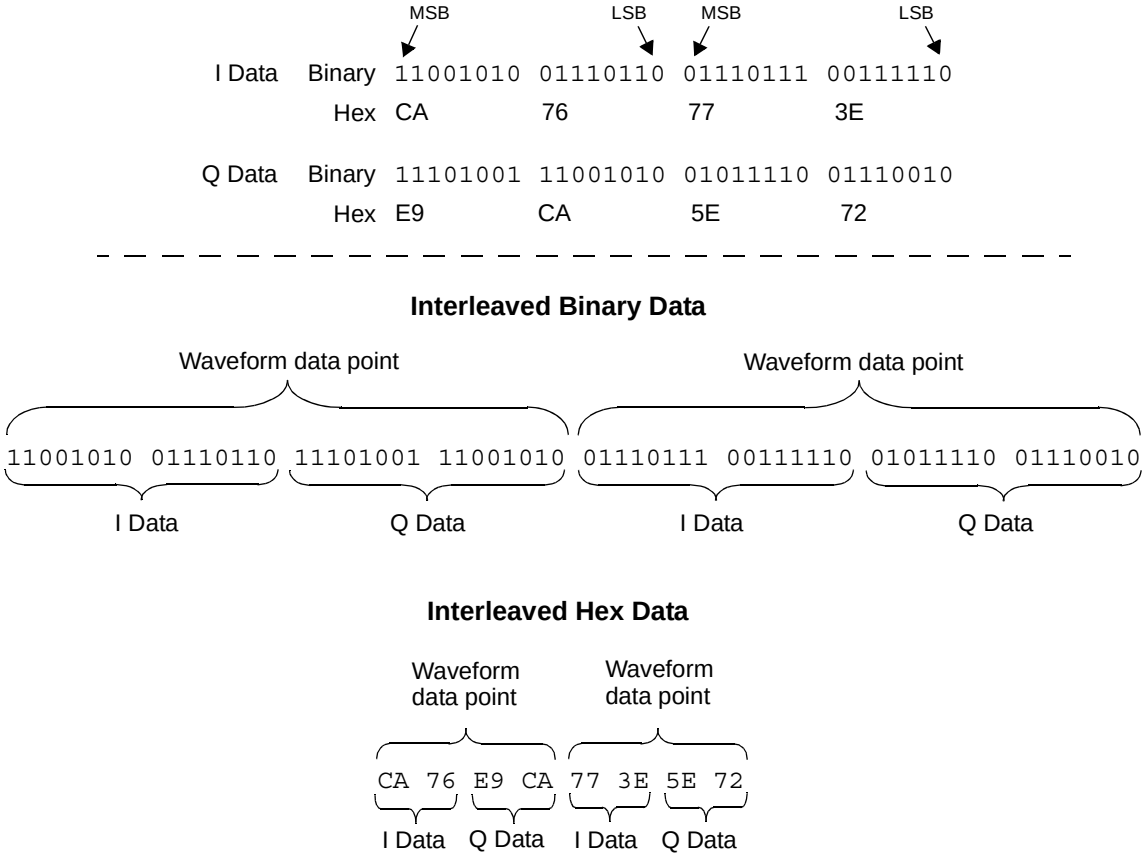
```
  01011100 10011110
+ 10100011 01100010
  00000000 00000000
```

I and Q Interleaving

When you create the waveform data, the I and Q data points typically reside in separate arrays or files. The signal generator requires a single I/Q file for waveform data playback. The process of interleaving creates a single array with alternating I and Q data points, with the Q data following the I data. This array is then downloaded to the signal generator as a binary file. The interleaved file comprises the waveform data points where each set of data points, one I data point and one Q data point, represents one I/Q waveform point.

NOTE The signal generator can accept separate I and Q files created for the earlier E443xB ESG models. For more information on downloading E443xB files, see "Downloading E443xB Signal Generator Files" on page 243.

The following figure illustrates interleaving I and Q data. Remember that it takes two bytes (16 bits) to represent one I or Q data point.



Waveform Structure

To play back waveforms, the signal generator uses data from the following three files:

- File header
- Marker file
- I/Q file

All three files have the same name, the name of the I/Q data file, but the signal generator stores each file in its respective directory (headers, markers, and waveform). For information on file extractions, see "Commands for Downloading and Extracting Waveform Data" on page 218.

File Header

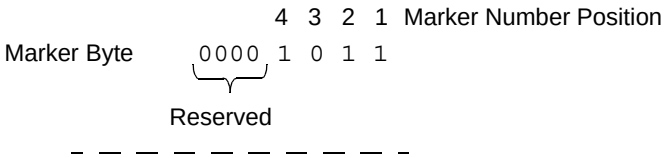
The file header contains settings for the ARB modulation format such as sample rate, marker polarity, I/Q modulation attenuator setting and so forth. When you create and download I/Q data, the signal generator automatically creates a file header with all saved parameters set to unspecified. With unspecified header settings, the waveform either uses the signal generator default settings, or if a waveform was previously played, the settings from that waveform. Ensure that you configure and save the file header settings for each waveform.

NOTE If you have no RF output when you play back a waveform, ensure that the marker RF blanking function has not been set for any of the markers. The marker RF blanking function is a header parameter that can be inadvertently set active for a marker by a previous waveform. To check for and turn RF blanking off manually, refer to "Configuring the Pulse/RF Blank (Agilent MXG)" on page 288 and "Configuring the Pulse/RF Blank (ESG/PSG)" on page 288.

Marker File

The marker file uses one byte per I/Q waveform point to set the state of the four markers either on (1) or off (0) for each I/Q point. When a marker is active (on), it provides an output trigger signal to the rear panel EVENT 1 connector (Marker 1 only) or and the AUX IO, event 2 connector pin (Markers 1, 2, 3, or 4), that corresponds to the active marker number. (For more information on active markers and their output trigger signal location, refer to your signal generator's *User's Guide*.) Because markers are set at each waveform point, the marker file contains the same number of bytes as there are waveform points. For example, for 200 waveform points, the marker file contains 200 bytes.

Although a marker point is one byte, the signal generator uses only bits 0–3 to configure the markers; bits 4–7 are reserved and set to zero. The following example shows a marker byte.



Example of Setting a Marker Byte

Binary 0000 0101

Hex 05

Sets markers 1 and 3 on for a waveform point

The following example shows a marker binary file (all values in hex) for a waveform with 200 points. Notice the first marker point, `0f`, shows all four markers on for only the first waveform point.

```
00000000: 0f 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
00000010: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
00000020: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
00000030: 01 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05
00000040: 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05
00000050: 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05
00000060: 05 05 05 05 04 04 04 04 04 04 04 04 04 04 04 04
00000070: 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04
00000080: 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04
00000090: 04 04 04 04 04 04 00 00 00 00 00 00 00 00 00 00
000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000c0: 00 00 00 00 00 00 00 00
```

`0f` = All markers on

`01` = Marker 1 on

`05` = Markers 1 and 3 on

`04` = Marker 3 on

`00` = No active markers

If you create your own marker file, its name must be the same as the waveform file. If you download I/Q data without a marker file, the signal generator automatically creates a marker file with all points set to zero. For more information on markers, see the *User's Guide*.

NOTE Downloading marker data using a file name that currently resides on the signal generator overwrites the existing marker file without affecting the I/Q (waveform) file. However, downloading just the I/Q data with the same file name as an existing I/Q file also overwrites the existing marker file setting all bits to zero.

I/Q File

The I/Q file contains the interleaved I and Q data points (signed 16–bit integers for each I and Q data point). Each I/Q point equals one waveform point. The signal generator stores the I/Q data in the waveform directory.

NOTE If you download I/Q data using a file name that currently resides on the signal generator, it also overwrites the existing marker file setting all bits to zero and the file header setting all parameters to unspecified.

Waveform

A waveform consists of samples. When you select a waveform for playback, the signal generator loads settings from the file header. When the ARB is on, it creates the waveform samples from the data in the marker and I/Q (waveform) files. The file header, while required, does not affect the number of bytes that compose a waveform sample. One sample contains five bytes:

I/Q Data		+	Marker Data	=	1 Waveform Sample
2 bytes I (16 bits)	2 bytes Q (16 bits)		1byte (8 bits) Bits 4–7 reserved—Bits 0–3 set		5 bytes

To create a waveform, the signal generator requires a minimum of 60 samples. To help minimize signal imperfections, use an even number of samples (for information on waveform continuity, see "Waveform Phase Continuity" on page 209). When you store waveforms, the signal generator saves changes to the waveform file, marker file, and file header.

Waveform Phase Continuity

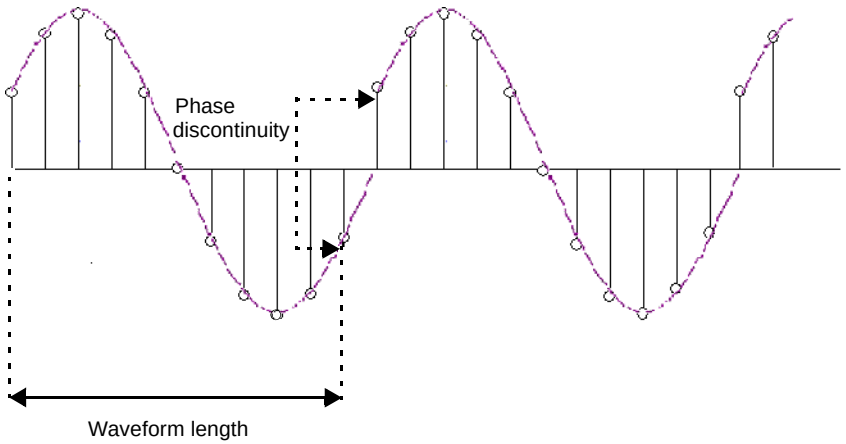
Phase Discontinuity, Distortion, and Spectral Regrowth

The most common arbitrary waveform generation use case is to play back a waveform that is finite in length and repeat it continuously. Although often overlooked, a phase discontinuity between the end of a waveform and the beginning of the next repetition can lead to periodic spectral regrowth and distortion.

For example, the sampled sinewave segment in the following figure may have been simulated in software or captured off the air and sampled. It is an accurate sinewave for the time period it occupies, however the waveform does not occupy an entire period of the sinewave or some multiple thereof. Therefore, when repeatedly playing back the waveform by an arbitrary waveform generator, a phase discontinuity is introduced at the transition point between the beginning and the end of the waveform.

Repetitions with abrupt phase changes result in high frequency spectral regrowth. In the case of playing back the sinewave samples, the phase discontinuity produces a noticeable increase in distortion components in addition to the line spectra normally representative of a single sinewave.

Sampled Sinewave with Phase Discontinuity



Avoiding Phase Discontinuities

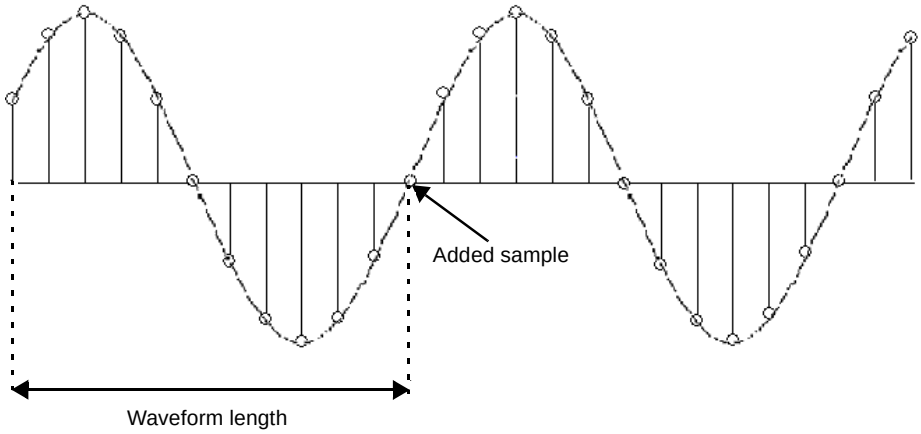
You can easily avoid phase discontinuities for periodic waveforms by simulating an integer number of cycles when you create your waveform segment.

NOTE If there are N samples in a complete cycle, only the first N–1 samples are stored in the waveform segment. Therefore, when continuously playing back the segment, the first and Nth waveform samples are always the same, preserving the periodicity of the waveform.

By adding off time at the beginning of the waveform and subtracting an equivalent amount of off time from the end of the waveform, you can address phase discontinuity for TDMA or pulsed periodic waveforms. Consequently, when the waveform repeats, the lack of signal present avoids the issue of phase discontinuity.

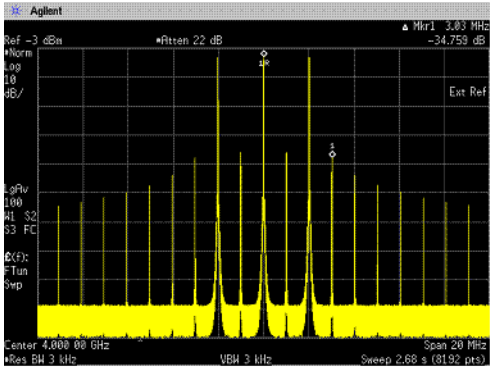
However, if the period of the waveform exceeds the waveform playback memory available in the arbitrary waveform generator, a periodic phase discontinuity could be unavoidable. N5110B Baseband Studio for Waveform Capture and Playback alleviates this concern because it does not rely on the signal generator waveform memory. It streams data either from the PC hard drive or the installed PCI card for N5110B enabling very large data streams. This eliminates any restrictions associated with waveform memory to correct for repetitive phase discontinuities. Only the memory capacity of the hard drive or the PCI card limits the waveform size.

Sampled Sinewave with No Discontinuity



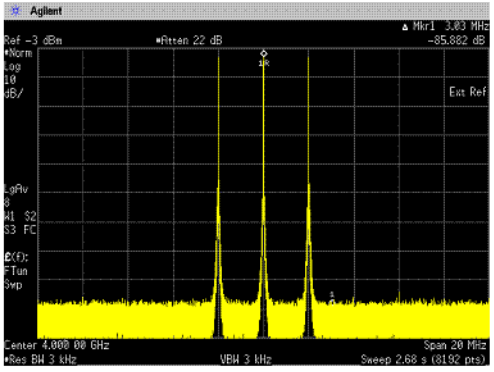
The following figures illustrate the influence a single sample can have. The generated 3–tone test signal requires 100 samples in the waveform to maintain periodicity for all three tones. The measurement on the left shows the effect of using the first 99 samples rather than all 100 samples. Notice all the distortion products (at levels up to –35 dBc) introduced in addition to the wanted 3–tone signal. The measurement on the right shows the same waveform using all 100 samples to maintain periodicity and avoid a phase discontinuity. Maintaining periodicity removes the distortion products.

Phase Discontinuity



3–tone – 20 MHz Bandwidth
Measured distortion = 35 dBc

Phase Continuity



3–tone – 20 MHz Bandwidth
Measured distortion = 86 dBc

Waveform Memory

The signal generator provides two types of memory, volatile and non–volatile. You can download files to either memory type.

Volatile — Random access memory that does not survive cycling of the signal generator power. This memory is commonly referred to as waveform memory (WFM1) or waveform playback memory. To play back waveforms, they must reside in volatile memory. The following file types share this memory:

Table 5-1 Signal Generators and Volatile Memory Types

Volatile Memory Type	Model of Signal Generator		
	N5162A, N5182A with Option 651, 652, or 654	E4438C with Option 001, 002, 601, or 602	E8267D Option 601 or 602
I/Q	x	x	x
Marker	x	x	x
File header	x	x	x
User PRAM	–	x	x

Non–volatile — Storage memory where files survive cycling the signal generator power. Files remain until overwritten or deleted. To play back waveforms after cycling the signal generator power, you must load waveforms from non–volatile waveform memory (NVWFM) to volatile waveform memory (WFM1). On the Agilent MXG the non–volatile memory is referred to as internal media and external media. The following file types share this memory:

Table 5-2 Signal Generators and Non–Volatile Memory Types

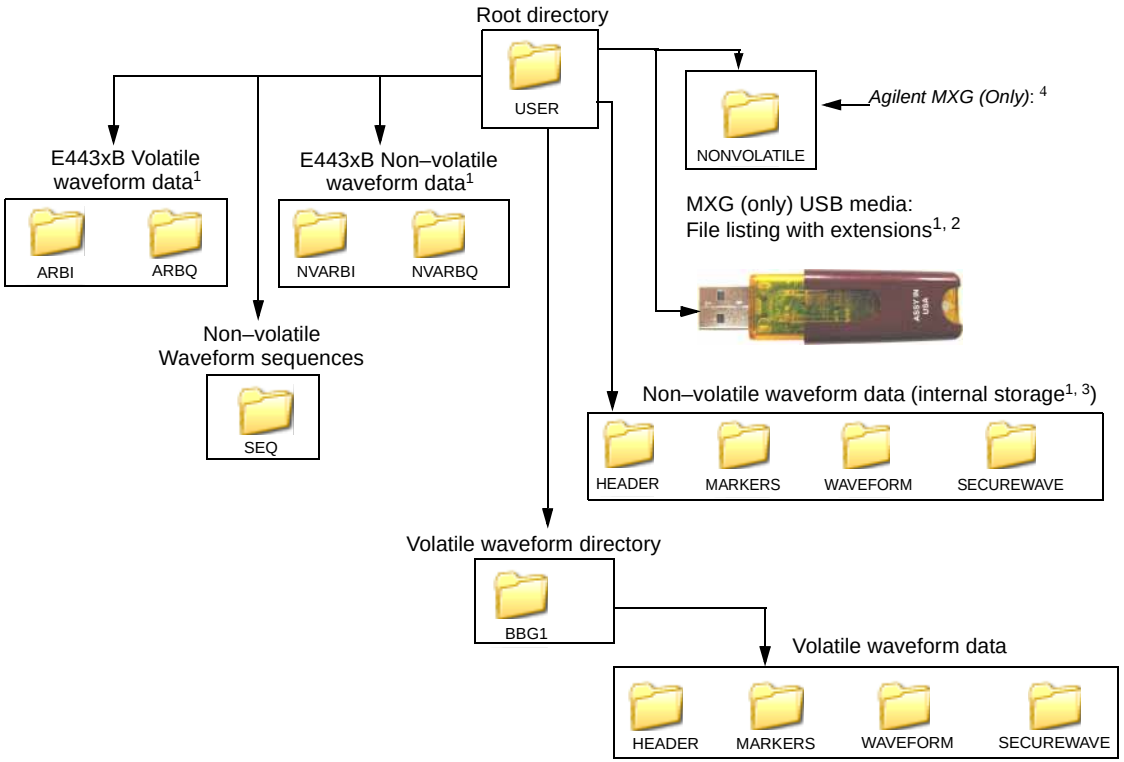
Non–Volatile Memory Type	Model of Signal Generator		
	N5162A, N5182A with Option 651, 652, or 654	E4438C with Option 001, 002, 601, or 602	E8267D Option 601 or 602
I/Q	x	x	x
Marker	x	x	x
File header	x	x	x
Sweep List	x	x	x
User Data	x	x	x

Table 5-2 Signal Generators and Non–Volatile Memory Types

Non–Volatile Memory Type	Model of Signal Generator		
	N5162A, N5182A with Option 651, 652, or 654	**E4438C with Option 001, 002, 601, or 602**	**E8267D Option 601 or 602**
User PRAM	–	x	x
Instrument State	x	x	x
Waveform Sequences (multiple I/Q files played together)	x	x	x

The following figure on Figure 5-1 on page 214 shows the locations within the signal generator for volatile and non–volatile waveform data.

Figure 5-1



[1]For information on using the E443xB directories, see "Downloading E443xB Signal Generator Files" on page 243.
[2]The Agilent MXG uses an optional "USB media" to store non–volatile waveform data.
[3]The Agilent MXG internal non–volatile memory is referred to as "internal storage".
[4]This NONVOLATILE directory shows the files with the same extensions as the USB media and is useful with ftp.

Memory Allocation

Volatile Memory

The signal generator allocates volatile memory in blocks of 1024 bytes. For example, a waveform file with 60 samples (the minimum number of samples) has 300 bytes (5 bytes per sample × 60 samples), but the signal generator allocates 1024 bytes of memory. If a waveform is too large to fit into 1024 bytes, the signal generator allocates additional memory in multiples of 1024 bytes. For example, the signal generator allocates 3072 bytes of memory for a waveform with 500 samples (2500 bytes).

3 x 1024 bytes = 3072 bytes of memory

As shown in the examples, waveforms can cause the signal generator to allocate more memory than what is actually used, which decreases the amount of available memory.

NOTE In the first block of data of volatile memory that is allocated for each waveform file, the file header requires 512 bytes (N5162A/82A) or 256 bytes (E4438C/E8267D).

Non–Volatile Memory (Agilent MXG)

NOTE If the Agilent MXG's external USB flash memory port is used, the USB flash memory can provide actual physical storage of non–volatile data in the SECUREWAVE directory versus the "virtual" only data.

ARB waveform encryption of proprietary information is supported on the external non–volatile USB flash memory.

To copy *unencrypted* data files from an external media (i.e. USB memory stick) for playing on a signal generator, the full filename extension is required (i.e. .MARKER, .HEADER, .WAVEFORM, etc.). For more information on unencrypted data, refer to "Commands for Downloading and Extracting Waveform Data" on page 218. For more information on how to work with files, refer to the *User's Guide*.

To copy *compatible licensed* encrypted data files (i.e. `.SECUREWAVE`) from an external media, download (copy) the files to the signal generator (refer to the *User's Guide* for information on how to work with files). When using the external media along with the signal generator's **Use as** or **Copy File to Instrument** softkey menus, encrypted data files can be automatically detected by the Agilent MXG, regardless of the suffix (e.g. .wfm, .wvfm, and *no suffix*, etc.). These various waveform files can be selected and played by the Agilent MXG. For more information on encrypted data, refer to "Commands for Downloading and Extracting Waveform Data" on page 218. When using the **Copy File to Instrument**, the signal generator prompts the user to select between BBG Memory and Internal Storage memories as locations to copy the files.

On the N5162A/82A, non–volatile files are stored on the non–volatile internal signal generator memory (internal storage) or to an USB media, if available.

The Agilent MXG non–volatile internal memory is allocated according to a Microsoft compatible file allocation table (FAT) file system. The Agilent MXG signal generator allocates non–volatile memory in clusters according to the drive size (see Table 5-3 on page 216). For example, referring to Table 5-3 on page 216, if the drive size is 15 MB and if the file is less than or equal to 4K bytes, the file uses only one 4 KB cluster of memory. For files larger than 4 KB, and with a drive size of 15 MB, the signal generator allocates additional memory in multiples of 4KB clusters. For example, a file that has 21,538 bytes consumes 6 memory clusters (24,000 bytes).

Microsoft is a registered trademark of Microsoft Corporation.

For more information on default cluster sizes for FAT file structures, refer to Table 5-3 on page 216 and to *http://support.microsoft.com/*.

Table 5-3 Drive Size (logical volume)

Drive Size (logical volume)	Cluster Size (Bytes) (Minimum Allocation Size)
0 MB – 15 MB	4K
16 MB – 127 MB	2K
128 MB – 255 MB	4K
256 MB – 511 MB	8K
512 MB – 1023 MB	16K
1024 MB – 2048 MB	32K
2048 MB – 4096 MB	64K
4096 MB – 8192 MB	128K
8192 MB – 16384 MB	256K

Non–Volatile Memory (ESG/PSG)

The ESG/PSG signal generators allocate non–volatile memory in blocks of 512 bytes. For files less than or equal to 512 bytes, the file uses only one block of memory. For files larger than 512 bytes, the signal generator allocates additional memory in multiples of 512 byte blocks. For example, a file that has 21,538 bytes consumes 43 memory blocks (22,016 bytes).

Memory Size

The amount of available memory, volatile and non–volatile, varies by option and the size of the other files that share the memory. When we refer to waveform files, we state the memory size in samples (one sample equals five bytes). The ESG and PSG baseband generator (BBG) options (001, 002, 601, or 602) and the Agilent MXG baseband generator (BBG) Option (651, 652, and 654) contain the waveform playback memory. Refer to Tables 5-4 on page 217 through Table 5-6 on page 218 for the maximum available memory.

Volatile and Non–Volatile Memory (N5162A/82A)

Table 5-4 N5162A/82A Volatile (BBG) and Non–Volatile (Internal Storage and USB Media) Memory

Volatile (BBG) Memory			Non–Volatile (Internal Storage and USB Media) Memory	
Option	**Size**		**Option**	**Size**
N5162A/82A[a]				
651/652/654 (BBG)	8 MSa (40 MB)		**Standard (N5182A)**	800 MSa (4 GB)[b]
019 (BBG)	64 MSa (320 MB)		**USB memory stick**	*user determined*

a. On the N5162A/82A, 512 bytes is reserved for each waveform's header file (i.e. The largest waveform that could be played with a N5162A/82A with Option 019 (320 MB) is: 320 MB – 512 bytes = 319,999,488 MB.)
b. For serial numbers <MY4818xxxx, US4818xxxx, and SG4818xxxx, the persistent memory value = 512 MB.

Volatile Memory and Non–Volatile Memory (E4438C and E8267D Only)

NOTE When considering volatile memory, it is not necessary to keep track of marker data, as this memory is consumed automatically and proportionally to the I/Q data created (i.e. 1 marker byte for every 4 bytes of I/Q data).

On the E4438C and E8267D, the fixed file system overhead on the signal generator is used to store directory information. When calculating the available volatile memory for waveform files it is important to consider the fixed file system overhead for the volatile memory of your signal generator.

Table 5-5 Fixed File System Overhead

Volatile (WFM1) Memory and Fixed File Overhead				
Option	**Size**	**Maximum Number of Files** *(MaxNumFiles)*	**Memory (Bytes) Used for Fixed File System Overhead**[a] *[16 + (44 x MaxNumFiles)]*	**Memory Available for Waveform Samples**
E4438C and E8267D				
001, 601 (BBG)	8 MSa (40 MB)	1024	46,080	*8,377,088 Samples*
002 (BBG)	32 MSa (160 MB)	4096	181,248	*33,509,120 Samples*
602 (BBG)	64 MSa (320 MB)	8192	361,472	*67,018,496 Samples*

a. The expression [16 + [44 x MaxNumFiles]) has been rounded up to nearest memory block (1024 bytes). (To find the I/Q waveform sample size, this resulting value needs to be divided by 4.)

Table 5-6 E4438C and E8267D Non–Volatile (NVWFM) Memory

Non–Volatile (NVWFM) Memory	
Option	**Size**
E4438C and E8267D	
Standard	3 MSa (15 MB)
005 (Hard disk)	1.2 GSa (6 GB)

Commands for Downloading and Extracting Waveform Data

You can download I/Q data, the associated file header, and marker file information (collectively called waveform data) into volatile or non–volatile memory. For information on waveform structure, see "Waveform Structure" on page 207.

The signal generator provides the option of downloading waveform data either for extraction or not for extraction. When you extract waveform data, the signal generator may require it to be read out in encrypted form. The SCPI download commands determine whether the waveform data is extractable.

If you use SCPI commands to download waveform data to be extracted later, you must use the `MEM:DATA:UNPRotected` command. If you use FTP commands, no special command syntax is necessary.

NOTE On the N5162A/82A, `:MEM:DATA` enables file extraction. On the N5162A/82A the `:MEM:DATA:UNPRotected` command is *not* required to enable file extraction. For more information, refer to the *SCPI Command Reference*.

You can download or extract waveform data created in any of the following ways:

- with signal simulation software, such as MATLAB or Agilent Advanced Design System (ADS)
- with advanced programming languages, such as C++, VB or VEE
- with Agilent Signal Studio software
- with the signal generator

Waveform Data Encryption

You can download encrypted waveform data extracted from one signal generator into another signal generator with the same option or software license for the modulation format. You can also extract encrypted waveform data created with software such as MATLAB or ADS, providing the data was downloaded to the signal generator using the proper command.

When you generate a waveform from the signal generator's internal ARB modulation format (ESG/PSG only), the resulting waveform data is automatically stored in volatile memory and is available for extraction as an encrypted file.

When you download an exported waveform using a Agilent Signal Studio software product, you can use the FTP process and the securewave directory or SCPI commands, to extract the encrypted file to the non–volatile memory on the signal generator. Refer to "File Transfer Methods" on page 219.

Encrypted I/Q Files and the Securewave Directory

The signal generator uses the `securewave` directory to perform file encryption (extraction) and decryption (downloads). The `securewave` directory is not an actual storage directory, but rather a portal for the encryption and decryption process. While the `securewave` directory contains file names, these are actually pointers to the true files located in signal generator memory (volatile or non–volatile). When you download an encrypted file, the `securewave` directory decrypts the file and unpackages the contents into its file header, I/Q data, and marker data. When you extract a file, the `securewave` directory packages the file header, I/Q data, and marker data and encrypts the waveform data file. When you extract the waveform file (I/Q data file), it includes the other two files, so there is no need to extract each one individually.

The signal generator uses the following `securewave` directory paths for file extractions and encrypted file downloads:

Volatile	*/user/bbg1/securewave/file_name* or *swfm:file_name*
Non–volatile	*/user/securewave* or *snvwfm1:file_name*

NOTE To extract files (other than user–created I/Q files) and to download encrypted files, you *must* use the `securewave` directory. If you attempt to extract previously downloaded encrypted files (including Signal Studio downloaded files or internally created signal generator files (ESG/PSG only)) *without* using the securewave directory, the signal generator generates an error and displays:
`ERROR: 221, Access Denied.`

Encrypted I/Q Files and the Securewave Directory (Agilent MXG)

NOTE Header parameters of files stored on the Agilent MXG's internal or USB media cannot be changed unless the file is copied to the volatile BBG memory. For more information on modifying header parameters, refer to the *User's Guide*.

When downloading encrypted files (`.SECUREWAVE`) from the USB media that have had the file suffix changed to something other than `.SECUREWAVE`, you must use the **Use As** or **Copy File to Instrument** menus to play an encrypted waveform file in the signal generator.

File Transfer Methods

- SCPI using VXI–11 (VMEbus Extensions for Instrumentation as defined in VXI–11)
- SCPI over the GPIB or RS 232
- SCPI with sockets LAN (using port 5025)
- File Transfer Protocol (FTP)

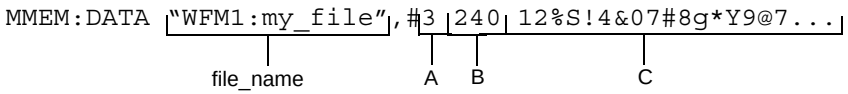
SCPI Command Line Structure

The signal generator expects to see waveform data as block data (binary files). The IEEE standard 488.2–1992 section 7.7.6 defines block data. The following example shows how to structure a SCPI command for downloading waveform data (#ABC represents the block data):

```
:MMEM:DATA "<file_name>",#ABC
```

"<file_name>"	the I/Q file name and file path within the signal generator
#	indicates the start of the data block
A	the number of decimal digits present in B
B	a decimal number specifying the number of data bytes to follow in C
C	the actual binary waveform data

The following example demonstrates this structure:



WFM1:	the file path
my_file	the I/Q file name as it will appear in the signal generator's memory catalog
#	indicates the start of the data block
3	B has three decimal digits
240	240 bytes of data to follow in C
12%S!4&07#8g*Y9@7...	the ASCII representation of some of the binary data downloaded to the signal generator, however not all ASCII values are printable

Commands and File Paths for Downloading and Extracting Waveform Data

---

NOTE Filenames should *not* exceed 23 characters.

You can download or extract waveform data using the commands and file paths in the following tables:

- Table 5-7, "Downloading Unencrypted Files for No Extraction (Extraction allowed on the Agilent MXG Only)," on page 221
- Table 5-8, "Downloading Encrypted Files for No Extraction (Extraction allowed on the Agilent MXG Only)," on page 221
- Table 5-9, "Downloading Unencrypted Files for Extraction," on page 221
- Table 5-11, "Downloading Encrypted Files for Extraction," on page 223
- Table 5-12, "Extracting Encrypted Waveform Data," on page 223

Table 5-7 Downloading Unencrypted Files for No Extraction (Extraction allowed on the Agilent MXG[a] Only)

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA "WFM1:<file_name>",<blockdata> MMEM:DATA "MKR1:<file_name>",<blockdata> MMEM:DATA "HDR1:<file_name>",<blockdata>
SCPI/volatile memory with full directory path	MMEM:DATA "user/bbg1/waveform/<file_name>",<blockdata> MMEM:DATA "user/bbg1/markers/<file_name>",<blockdata> MMEM:DATA "user/bbg1/header/<file_name>",<blockdata>
SCPI/non–volatile memory	MMEM:DATA "NVWFM:<file_name>",<blockdata> MMEM:DATA "NVMKR:<file_name>",<blockdata> MMEM:DATA "NVHDR:<file_name>",<blockdata>
SCPI/non–volatile memory with full directory path	MMEM:DATA /user/waveform/<file_name>",<blockdata> MMEM:DATA /user/markers/<file_name>",<blockdata> MMEM:DATA /user/header/<file_name>",<blockdata>

a. Refer to note on page 218.

Table 5-8 Downloading Encrypted Files for No Extraction (Extraction allowed on the Agilent MXG[a] Only)

Download Method /Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA "user/bbg1/securewave/<file_name>",<blockdata> MMEM:DATA "SWFM1:<file_name>",<blockdata> MMEM:DATA "file_name@SWFM1",<blockdata>
SCPI/non–volatile memory	MMEM:DATA "user/securewave/<file_name>",<blockdata> MMEM:DATA "SNVWFM:<file_name>",<blockdata> MMEM:DATA "file_name@SNVWFM",<blockdata>

a. Refer to note on page 218.

Table 5-9 Downloading Unencrypted Files for Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory[a]	MEM:DATA:UNPRotected "/user/bbg1/waveform/file_name",<blockdata> MEM:DATA:UNPRotected "/user/bbg1/markers/file_name",<blockdata> MEM:DATA:UNPRotected "/user/bbg1/header/file_name",<blockdata> MEM:DATA:UNPRotected "WFM1:file_name",<blockdata> MEM:DATA:UNPRotected "MKR1:file_name",<blockdata> MEM:DATA:UNPRotected "HDR1:file_name",<blockdata> MEM:DATA:UNPRotected "file_name@WFM1",<blockdata> MEM:DATA:UNPRotected "file_name@MKR1",<blockdata> MEM:DATA:UNPRotected "file_name@HDR1",<blockdata>

Table 5-9 Downloading Unencrypted Files for Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/non–volatile memory[a]	MEM:DATA:UNPRotected "/user/waveform/file_name",<blockdata> MEM:DATA:UNPRotected "/user/markers/file_name",<blockdata> MEM:DATA:UNPRotected "/user/header/file_name",<blockdata> MEM:DATA:UNPRotected "NVWFM:file_name",<blockdata> MEM:DATA:UNPRotected "NVMKR:file_name",<blockdata> MEM:DATA:UNPRotected "NVHDR:file_name",<blockdata> MEM:DATA:UNPRotected "file_name@NVWFM",<blockdata> MEM:DATA:UNPRotected "file_name@NVMKR",<blockdata> MEM:DATA:UNPRotected "file_name@NVHDR",<blockdata>
FTP/volatile memory[b]	put <file_name> /user/bbg1/waveform/<file_name> put <file_name> /user/bbg1/markers/<file_name> put <file_name> /user/bbg1/header/<file_name>
FTP/non–volatile memory[b]	put <file_name> /user/waveform/<file_name> put <file_name> /user/markers/<file_name> put <file_name> /user/header/<file_name>

a. On the N5162A/82A the :MEM:DATA:UNPRotected command is *not* required to be able to extract files (i.e. use :MEM:DATA). For more information, refer to the *SCPI Command Reference*.
b. See "FTP Procedures" on page 224.

Table 5-10 Extracting Unencrypted I/Q Data

Download Method/Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA? "/user/bbg1/waveform/<file_name>" MMEM:DATA? "WFM1:<file_name>" MMEM:DATA? "<file_name>@WFM1"
SCPI/non–volatile memory	MMEM:DATA? "/user/waveform/<file_name>" MMEM:DATA? "NVWFM:<file_name>" MMEM:DATA? "<file_name>@NVWFM"
FTP/volatile memory[a]	get /user/bbg1/waveform/<file_name> get /user/bbg1/markers/<file_name> get /user/bbg1/header/<file_name>
FTP/non–volatile memory[a]	get /user/waveform/<file_name> get /user/markers/<file_name> get /user/header/<file_name>

a. See "FTP Procedures" on page 224.

Table 5-11 Downloading Encrypted Files for Extraction

Download Method/Memory Type	Command Syntax Options
SCPI/volatile[a] memory	MEM:DATA:UNPRotected "/user/bbg1/securewave/file_name",<blockdata> MEM:DATA:UNPRotected "SWFM1:file_name",<blockdata> MEM:DATA:UNPRotected "file_name@SWFM1",<blockdata>
SCPI/non–volatile memory[a]	MEM:DATA:UNPRotected "/user/securewave/file_name",<blockdata> MEM:DATA:UNPRotected "SNVWFM:file_name",<blockdata> MEM:DATA:UNPRotected "file_name@SNVWFM",<blockdata>
FTP/volatile memory[b]	put <file_name> /user/bbg1/securewave/<file_name>
FTP/non–volatile memory[b]	put <file_name> /user/securewave/<file_name>

a. On the N5162A/82A the :MEM:DATA:UNPRotected command is *not* required to be able to extract files (i.e. use :MEM:DATA). For more information, refer to the *SCPI Command Reference*.
b. See "FTP Procedures" on page 224.

Table 5-12 Extracting Encrypted Waveform Data

Download Method/Memory Type	Command Syntax Options
SCPI/volatile memory	MMEM:DATA? "/user/bbg1/securewave/file_name" MMEM:DATA? "SWFM1:file_name" MMEM:DATA? "file_name@SWFM1"
SCPI/non–volatile memory	MMEM:DATA? "/user/securewave/file_name" MMEM:DATA? "SNVWFM:file_name" MMEM:DATA? "file_name@SNVWFM"
FTP/volatile memory[a]	get /user/bbg1/securewave/<file_name>
FTP/non–volatile memory[a]	get /user/securewave/<file_name>

a. See "FTP Procedures" on page 224.

FTP Procedures

CAUTION Avoid using the *OPC? or *WAI commands to verify that the FTP process has been completed. These commands can potentially hang up due to the processing of other SCPI parser operations. Refer to the *SCPI Command Reference*.

NOTE If you are remotely FTPing files and need to verify the completion of the FTP process, then query the instrument by using SCPI commands such as: ':MEM:DATA:', ':MEM:CAT', '*STB?', 'FREQ?', '*IDN?', 'OUTP:STAT?'. Refer to the *SCPI Command Reference*.

There are three ways to FTP files:

- use Microsoft's® Internet Explorer FTP feature
- use the PC's or UNIX command window
- use the signal generator's internal web server following the firmware requirements in the table below

Signal Generator	Firmware Version (Required for Web Server Compatibility)
N516xA[a], N518xA	All
E44x8C	≥ C.03.10
E82x7D, E8663B	All

a. The N5161A and N5162A require firmware versions A.0140 or newer.

Using Microsoft's Internet Explorer

1. Enter the signal generator's hostname or IP address as part of the FTP URL.

   ftp://<host name> or

   ftp://<IP address>

2. Press **Enter** on the keyboard or **Go** from the Internet Explorer window.

   The signal generator files appear in the Internet Explorer window.

3. Drag and drop files between the PC and the Internet Explorer window

Using the Command Window (PC or UNIX)

This procedure downloads to non–volatile memory. To download to volatile memory, change the file path.

CAUTION Get and Put commands write over existing files by the same name in destination directories. Remember to change remote and local filenames to avoid the loss of data.

NOTE If a filename has a space, quotations are required around the filename.

Always transfer the waveform file before transferring the marker file.

For additional information on FTP commands, refer to the operating system's command window help.

1. From the PC command prompt or UNIX command line, change to the destination directory for the file you intend to download.
2. From the PC command prompt or UNIX command line, type `ftp <instrument name>`. Where `instrument name` is the signal generator's hostname or IP address.
3. At the `User:` prompt in the ftp window, press **Enter** (no entry is required).
4. At the `Password:` prompt in the ftp window, press **Enter** (no entry is required).
5. At the `ftp` prompt, either **put** a file or **get** a file:

   To put a file, type:

   `put <file_name> /user/waveform/<file_name1>`

   where `<file_name>` is the name of the file to download and `<file_name1>` is the name designator for the signal generator's `/user/waveform/` directory.

   If `<filename1>` is unspecified, ftp uses the specified `<file_name>` to name `<file_name1>`.

   - If a marker file is associated with the data file, use the following command to download it to the signal generator:
     `put <marker file_name> /user/markers/<file_name1>`

     where <marker file_name> is the name of the file to download and <file_name1> is the name designator for the file in the signal generator's `/user/markers/` directory. Marker files and the associated I/Q waveform data have the same name.

     For more examples of put command usage refer to Table 5-13.

Table 5-13 Put Command Examples

Command Results	Local	Remote	Notes
Incorrect	`put <filename.wfm>` `put <filename.mkr>`	`/user/waveform/<filename1.wfm>` `/user/marker/<filename1.mkr>`	Produces two separate and incompatible files.

Table 5-13 Put Command Examples

Command Results	Local	Remote	Notes
Correct	`put <filename.wfm>` `put <filename.mkr>`	`/user/waveform/<filename1>` `/user/marker/<filename1>`	Creates a waveform file and a compatible marker file.

To get a file, type:

`get /user/waveform/<file_name1> <file_name>`

where `<file_name1>` is the file to download from the signal generator's `/user/waveform/` directory and `<file_name>` is the name designator for the local PC/UNIX.

- If a marker file is associated with the data file, use the following command to download it to the local PC/UNIX directory:
  `get /user/markers/<file_name1> <marker file_name>`

  where `<marker file_name1>` is the name of the marker file to download from the signal generator's `/user/markers/` directory and `<marker file_name>` is the name of the file to be downloaded to the local PC/UNIX.

  For more examples of get command usage refer to Table 5-14.

Table 5-14 Get Command Examples

Command Results	Local	Remote	Notes
Incorrect	`get /user/waveform/file` `get /user/marker/file`	`file1` `file1`	Results in file1 containing only the marker data.
Correct	`get /user/waveform/file` `get /user/marker/file`	`file1.wfm` `file1.mkr`	Creates a waveform file and a compatible marker file. It is easier to keep files associated by varying the extenders.

6. At the `ftp` prompt, type: `bye`
7. At the command prompt, type: `exit`

Using the Signal Generator's Internal Web Server

1. Enter the signal generator's hostname or IP address in the URL.

   http://<host name> or <IP address>

2. Click the **Signal Generator FTP Access** button located on the left side of the window.

   The signal generator files appear in the web browser's window.

3. Drag and drop files between the PC and the browser's window

For more information on the web server feature, see Chapter 1.

Creating Waveform Data

This section examines the C++ code algorithm for creating I/Q waveform data by breaking the programming example into functional parts and explaining the code in generic terms. This is done to help you understand the code algorithm in creating the I and Q data, so you can leverage the concept into your programming environment. The *SCPI Command Reference*, contains information on how to use SCPI commands to define the markers (polarity, routing, and other marker settings). If you do not need this level of detail, you can find the complete programming examples in "Programming Examples" on page 246.

You can use various programming environments to create ARB waveform data. Generally there are two types:

- **Simulation software**– this includes MATLAB, Agilent Technologies EESof Advanced Design System (ADS), Signal Processing WorkSystem (SPW), and so forth.
- **Advanced programming languages**–this includes, C++, VB, VEE, MS Visual Studio.Net, Labview, and so forth.

No matter which programming environment you use to create the waveform data, make sure that the data conforms to the data requirements shown on page 199. To learn about I/Q data for the signal generator, see "Understanding Waveform Data" on page 199.

Code Algorithm

This section uses code from the C++ programming example "Importing, Byte Swapping, Interleaving, and Downloading I and Q Data–Big and Little Endian Order" on page 262 to demonstrate how to create and scale waveform data.

There are three steps in the process of creating an I/Q waveform:

1. Create the I and Q data.
2. Save the I and Q data to a text file for review.
3. Interleave the I and Q data to make an I/Q file, and swap the byte order for little–endian platforms.

For information on downloading I/Q waveform data to a signal generator, refer to "Commands and File Paths for Downloading and Extracting Waveform Data" on page 220 and "Downloading Waveform Data" on page 233.

1. Create I and Q data.

The following lines of code create scaled I and Q data for a sine wave. The I data consists of one

period of a sine wave and the Q data consists of one period of a cosine wave.

Line	Code—Create I and Q data
1	`const int NUMSAMPLES=500;`
2	`main(int argc, char* argv[]);`
3	`{`
4	`short idata[NUMSAMPLES];`
5	`short qdata[NUMSAMPLES];`
6	`int numsamples = NUMSAMPLES;`
7	`for(int index=0; index<numsamples; index++);`
8	`{`
9	`idata[index]=23000 * sin((2*3.14*index)/numsamples);`
10	`qdata[index]=23000 * cos((2*3.14*index)/numsamples);`
11	`}`

Line	Code Description—Create I and Q data
1	Define the number of waveform points. Note that the maximum number of waveform points that you can set is based on the amount of available memory in the signal generator. For more information on signal generator memory, refer to "Waveform Memory" on page 212.
2	Define the main function in C++.
4	Create an array to hold the generated I values. The array length equals the number of the waveform points. Note that we define the array as type *short*, which represents a 16–bit signed integer in most C++ compilers.
5	Create an array to hold the generated Q values (signed 16–bit integers).
6	Define and set a temporary variable, which is used to calculate the I and Q values.

Line	Code Description—Create I and Q data
7–11	Create a loop to do the following: • Generate and scale the I data (DAC values). This example uses a simple sine equation, where 2*3.14 equals one waveform cycle. Change the equation to fit your application. — The array pointer, *index*, increments from 0–499, creating 500 I data points over one period of the sine waveform. — Set the scale of the DAC values in the range of –32768 to 32767, where the values –32768 and 32767 equal full scale negative and positive respectively. This example uses 23000 as the multiplier, resulting in approximately 70% scaling. For more information on scaling, see "Scaling DAC Values" on page 203. **NOTE** The signal generator comes from the factory with I/Q scaling set to 70%. If you reduce the DAC input values, ensure that you set the signal generator scaling (`:RADio:ARB:RSCaling`) to an appropriate setting that accounts for the reduced values. • Generate and scale the Q data (DAC value). This example uses a simple cosine equation, where 2*3.14 equals one waveform cycle. Change the equation to fit your application. — The array pointer, *index*, increments from 0–499, creating 500 Q data points over one period of the cosine waveform. — Set the scale of the DAC values in the range of –32767 to 32768, where the values –32767 and 32768 equal full scale negative and positive respectively. This example uses 23000 as the multiplier, resulting in approximately 70% scaling. For more information on scaling, see "Scaling DAC Values" on page 203.

2. Save the I/Q data to a text file to review.

The following lines of code export the I and Q data to a text file for validation. After exporting the data, open the file using Microsoft Excel or a similar spreadsheet program, and verify that the I and Q data are correct.

Line **Code Description—Saving the I/Q Data to a Text File**

```
12  char *ofile = "c:\\temp\\iq.txt";
13  FILE *outfile = fopen(ofile, "w");
14  if (outfile==NULL) perror ("Error opening file to write");
15  for(index=0; index<numsamples; index++)
16  {
17     fprintf(outfile, "%d, %d\n", idata[index], qdata[index]);
18  }
19  fclose(outfile);
```

Line	Code Description—Saving the I/Q Data to a Text File
12	Set the absolute path of a text file to a character variable. In this example, *iq.txt is the file name and *ofile is the variable name.* For the file path, some operating systems may not use the drive prefix ('c:' in this example), or may require only a single forward slash (/), or both ("/*temp/iq.txt*")
13	Open the text file in *write* format.
14	If the text file does not open, print an error message.
15–18	Create a loop that prints the array of generated I and Q data samples to the text file.
19	Close the text file.

3. Interleave the I and Q data, and byte swap if using little endian order.

This step has two sets of code:

- Interleaving and byte swapping I and Q data for little endian order
- Interleaving I and Q data for big endian order

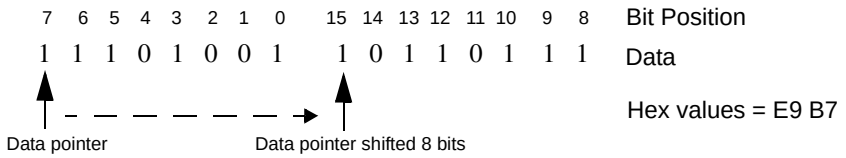
For more information on byte order, see "Little Endian and Big Endian (Byte Order)" on page 200.

Line	Code—Interleaving and Byte Swapping for Little Endian Order
20	`char iqbuffer[NUMSAMPLES*4];`
21	`for(index=0; index<numsamples; index++)`
22	`{`
23	`   short ivalue = idata[index];`
24	`   short qvalue = qdata[index];`
25	`   iqbuffer[index*4]   = (ivalue >> 8) & 0xFF;`
26	`   iqbuffer[index*4+1] = ivalue & 0xFF;`
27	`   iqbuffer[index*4+2] = (qvalue >> 8) & 0xFF;`
28	`   iqbuffer[index*4+3] = qvalue & 0xFF;`
29	`}`
30	`return 0;`

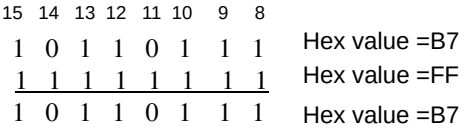
Line	Code Description—Interleaving and Byte Swapping for Little Endian Order
20	Define a character array to store the interleaved I and Q data. The character array makes byte swapping easier, since each array location accepts only 8 bits (1 byte). The array size increases by four times to accommodate two bytes of I data and two bytes of Q data.
21–29	Create a loop to do the following: • Save the current I data array value to a variable. **NOTE** In rare instances, a compiler may define *short* as larger than 16 bits. If this condition exists, replace *short* with the appropriate object or label that defines a 16–bit integer. • Save the current Q data array value to a variable. • Swap the low bytes (bits 0–7) of the data with the high bytes of the data (done for both

Line	Code Description—Interleaving and Byte Swapping for Little Endian Order
21–29	the I and Q data), and interleave the I and Q data. — shift the data pointer right 8 bits to the beginning of the high byte (*ivalue* >> *8*)

Little Endian Order

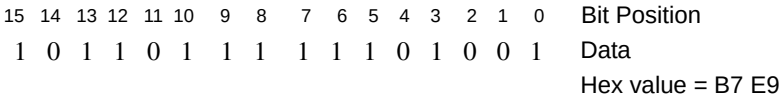

7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8
Bit Position
1 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1
Data
Hex values = E9 B7
Data pointer
Data pointer shifted 8 bits

— *AND* (boolean) the high I byte with 0xFF to make the high I byte the value to store in the IQ array—(*ivalue* >> *8*) *& 0xFF*



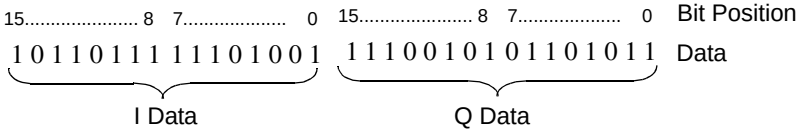
— *AND* (boolean) the low I byte with 0xFF (*ivalue & 0xFF*) to make the low I byte the value to store in the I/Q array location just after the high byte [*index* * 4 + 1]

I Data in I/Q Array after Byte Swap (Big Endian Order)



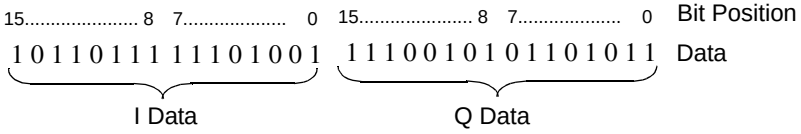
— Swap the Q byte order within the same loop. Notice that the I and Q data interleave with each loop cycle. This is due to the I/Q array shifting by one location for each I and Q operation [*index* * 4 + n].

Interleaved I/Q Array in Big Endian Order



Line **Code—Interleaving I and Q data for Big Endian Order**

```
20  short iqbuffer[NUMSAMPLES*2];
21  for(index=0; index<numsamples; index++)
22  {
23  iqbuffer[index*2]   = idata[index];
24  iqbuffer[index*2+1] = qdata[index];
25  }
26  return 0;
```

Line	Code Description—Interleaving I and Q data for Big Endian Order
20	Define a 16–bit integer (short) array to store the interleaved I and Q data. The array size increases by two times to accommodate two bytes of I data and two bytes of Q data. **NOTE** In rare instances, a compiler may define *short* as larger than 16 bits. If this condition exists, replace *short* with the appropriate object or label that defines a 16–bit integer.
21–25	Create a loop to do the following: • Store the I data values to the I/Q array location [*index**2]. • Store the Q data values to the I/Q array location [*index**2+1]. **Interleaved I/Q Array in Big Endian Order** 

To download the data created in the above example, see "Using Advanced Programming Languages" on page 236.

Downloading Waveform Data

This section examines methods of downloading I/Q waveform data created in MATLAB (a simulation software) and C++ (an advanced programming language). For more information on simulation and advanced programming environments, see "Creating Waveform Data" on page 227.

To download data from simulation software environments, it is typically easier to use one of the free download utilities (described on page 242), because simulation software usually saves the data to a file. In MATLAB however, you can either save data to a .mat file or create a complex array. To facilitate downloading a MATLAB complex data array, Agilent created the Agilent Waveform Download Assistant (one of the free download utilities), which downloads the complex data array from within the MATLAB environment. This section shows how to use the Waveform Download Assistant.

For advanced programming languages, this section closely examines the code algorithm for downloading I/Q waveform data by breaking the programming examples into functional parts and explaining the code in generic terms. This is done to help you understand the code algorithm in downloading the interleaved I/Q data, so you can leverage the concept into your programming environment. While not discussed in this section, you may also save the data to a binary file and use one of the download utilities to download the waveform data (see "Using the Download Utilities" on page 242).

If you do not need the level of detail this section provides, you can find complete programming examples in "Programming Examples" on page 246. Prior to downloading the I/Q data, ensure that it conforms to the data requirements shown on page 199. To learn about I/Q data for the signal generator, see "Understanding Waveform Data" on page 199. For creating waveform data, see "Creating Waveform Data" on page 227.

NOTE To avoid overwriting the current waveform in volatile memory, before downloading files into volatile memory (`WFM1`), change the file name or turn off the ARB. For more information, on manually turning off the ARB, refer to the *User's Guide*.

To turn off the ARB remotely, send: `:SOURce:RADio:ARB:STATe OFF`.

Using Simulation Software

This procedure uses a complex data array created in MATLAB and uses the Agilent Waveform Download Assistant to download the data. To obtain the Agilent Waveform Download Assistant, see "Using the Download Utilities" on page 242.

There are two steps in the process of downloading an I/Q waveform:

1. Open a connection session.
2. Download the I/Q data.

1. Open a connection session with the signal generator.

The following code establishes a LAN connection with the signal generator, sends the IEEE SCPI command `*idn?`, and if the connection fails, displays an error message.

Line	Code—Open a Connection Session
1	`io = agt_newconnection('tcpip','IP address');`
	`%io = agt_newconnection('gpib',<primary address>,<secondary address>);`
2	`[status,status_description,query_result] = agt_query(io,'*idn?');`
3	`if status == -1`
4	`display 'fail to connect to the signal generator';`
5	`end;`

Line	Code Description—Open a Connection Session with the Signal Generator
1	Sets up a structure (indicated above by *io*) used by subsequent function calls to establish a LAN connection to the signal generator. • *agt_newconnection()* is the function of Agilent Waveform Download Assistant used in MATLAB to build a connection to the signal generator. • If you are using GPIB to connect to the signal generator, provide the board, primary address, and secondary address: *io = agt_newconnection('gpib',0,19);* Change the GPIB address based on your instrument setting.
2	Send a query to the signal generator to verify the connection. • *agt_query()* is an Agilent Waveform Download Assistant function that sends a query to the signal generator. • If signal generator receives the query `*idn?`, *status* returns zero and *query_result* returns the signal generator's model number, serial number, and firmware version.
3–5	If the query fails, display a message.

2. Download the I/Q data

The following code downloads the generated waveform data to the signal generator, and if the download fails, displays a message.

Line	Code—Download the I/Q data
6	`[status, status_description] = agt_waveformload(io, IQwave, 'waveformfile1', 2000, 'no_play','norm_scale');`
7	`if status == -1`
8	`display 'fail to download to the signal generator';`
9	`end;`

Line	Code Description—Download the I/Q data
6	Download the I/Q waveform data to the signal generator by using the function call (*agt_waveformload*) from the Agilent Waveform Download Assistant. Some of the arguments are optional as indicated below, but if one is used, you must use all arguments previous to the one you require. Notice that with this function, you can perform the following actions: • download complex I/Q data • name the file (optional argument) • set the sample rate (optional argument) If you do not set a value, the signal generator uses its preset value of 125 MHz (N5162A/82A) or 100 MHz (E4438C/E8267D), or if a waveform was previously play, the value from that waveform. • start or not start waveform playback after downloading the data (optional argument) Use either the argument *play* or the argument *no_play*. • whether to normalize and scale the I/Q data (optional argument) If you normalize and scale the data within the body of the code, then use *no_normscale*, but if you need to normalize and scale the data, use *norm_scale*. This normalizes the waveform data to the DAC values and then scales the data to 70% of the DAC values. • download marker data (optional argument) If there is no marker data, the signal generator creates a default marker file, all marker set to zero. To verify the waveform data download, see "Loading, Playing, and Verifying a Downloaded Waveform" on page 239.
7–9	If the download fails, display an error message.

Using Advanced Programming Languages

This procedure uses code from the C++ programming example "Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order" on page 262.

For information on creating I/Q waveform data, refer to "Creating Waveform Data" on page 227.

There are two steps in the process of downloading an I/Q waveform:

1. Open a connection session.
2. Download the I/Q data.

1. Open a connection session with the signal generator.

The following code establishes a LAN connection with the signal generator or prints an error message if the session is not opened successfully.

Line	Code Description—Open a Connection Session
1	`char* instOpenString ="lan[hostname or IP address]";`
	`//char* instOpenString ="gpib<primary addr>,<secondary addr>";`
2	`INST id=iopen(instOpenString);`
3	`if (!id)`
4	`{`
5	`   fprintf(stderr, "iopen failed (%s)\n", instOpenString);`
6	`   return -1;`
7	`}`

Line	Code Description—Open a Connection Session
1	Assign the signal generator's LAN hostname, IP address, or GPIB address to a character string. • This example uses the Agilent IO library's *iopen()* SICL function to establish a LAN connection with the signal generator. The input argument, *lan[hostname or IP address]* contains the device, interface, or commander address. Change it to your signal generator host name or just set it to the IP address used by your signal generator. For example: "*lan[999.137.240.9]*" • If you are using GPIB to connect to the signal generator, use the commented line in place of the first line. Insert the GPIB address based on your instrument setting, for example "*gpib0,19*". • For the detailed information about the parameters of the SICL function *iopen()*, refer to the online "*Agilent SICL User's Guide for Windows*."
2	Open a connection session with the signal generator to download the generated I/Q data. The SICL function *iopen()* is from the Agilent IO library and creates a session that returns an identifier to *id*. • If *iopen()* succeeds in establishing a connection, the function returns a valid session *id*. The valid session *id* is not viewable, and can only be used by other SICL functions. • If *iopen()* generates an error before making the connection, the session identifier is always set to zero. This occurs if the connection fails. • To use this function in C++, you must include the standard header #include <sicl.h> before the main() function.
3–7	If *id* = 0, the program prints out the error message and exits the program.

2. Download the I/Q data.

The following code sends the SCPI command and downloads the generated waveform data to the signal generator.

Line CodeDescription—Download the I/Q Data

```
8   int bytesToSend;
9   bytesToSend = numsamples*4;
10  char s[20];
11  char cmd[200];
12  sprintf(s, "%d", bytesToSend);
13  sprintf(cmd, ":MEM:DATA \"WFM1:FILE1\", #%d%d", strlen(s), bytesToSend);
    iwrite(id, cmd, strlen(cmd), 0, 0);
14  iwrite(id, iqbuffer, bytesToSend, 0, 0);
15  iwrite(id, "\n", 1, 1, 0);
16
```

Line	Code Description—Download the I/Q data
8	Define an integer variable (*bytesToSend*) to store the number of bytes to send to the signal generator.

Line	Code Description—Download the I/Q data
9	Calculate the total number of bytes, and store the value in the integer variable defined in line 8. In this code, *numsamples* contains the number of waveform points, not the number of bytes. Because it takes four bytes of data, two I bytes and two Q bytes, to create one waveform point, we have to multiply *numsamples* by four. This is shown in the following example: numsamples = 500 waveform points numsamples × 4 = 2000 (four bytes per point) bytesToSend = 2000 (numsamples × 4) For information on setting the number of waveform points, see "1. Create I and Q data." on page 227.
10	Create a string large enough to hold the *bytesToSend* value as characters. In this code, string *s* is set to 20 bytes (20 characters—one character equals one byte)
11	Create a string and set its length (*cmd*[200]) to hold the SCPI command syntax and parameters. In this code, we define the string length as 200 bytes (200 characters).
12	Store the value of *bytesToSend* in string *s*. For example, if bytesToSend = 2000; *s* = "2000" *sprintf()* is a standard function in C++, which writes string data to a string variable.
13	Store the SCPI command syntax and parameters in the string *cmd*. The SCPI command prepares the signal generator to accept the data. • *strlen()* is a standard function in C++, which returns length of a string. • If *bytesToSend* = 2000, then *s* = "2000", *strlen(s)* = 4, so *cmd* = :MEM:DATA "WFM1:FILE1\" #42000.
14	Send the SCPI command stored in the string *cmd* to the signal generator, which is represented by the session *id*. • *iwrite()* is a SICL function in Agilent IO library, which writes the data (block data) specified in the string *cmd* to the signal generator (*id*). • The third argument of *iwrite(), strlen(cmd),* informs the signal generator of the number of bytes in the command string. The signal generator parses the string to determine the number of I/Q data bytes it expects to receive. • The fourth argument of iwrite(), 0, means there is no END of file indicator for the string. This lets the session remain open, so the program can download the I/Q data.

Line	Code Description—Download the I/Q data
15	Send the generated waveform data stored in the I/Q array *(iqbuffer)* to the signal generator. • *iwrite()* sends the data specified in *iqbuffer* to the signal generator (session identifier specified in *id*). • The third argument of iwrite(), *bytesToSend*, contains the length of the *iqbuffer* in bytes. In this example, it is 2000. • The fourth argument of *iwrite()*, 0, means there is no END of file indicator in the data. In many programming languages, there are two methods to send SCPI commands and data: — Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data. — Method 2 where the program sends a fixed number of bytes and ignores any zeros in the data. This is the method used in our program. For your programming language, you must find and use the equivalent of method two. Otherwise you may only achieve a partial download of the I and Q data.
16	Send the terminating carriage (\n) as the last byte of the waveform data. • *iwrite()* writes the data "\n" to the signal generator (session identifier specified in *id).* • The third argument of *iwrite()*, 1, sends one byte to the signal generator. • The fourth argument of *iwrite(),* 1, is the END of file indicator, which the program uses to terminate the data download. To verify the waveform data download, see "Loading, Playing, and Verifying a Downloaded Waveform" on page 239.

Loading, Playing, and Verifying a Downloaded Waveform

The following procedures show how to perform the steps using SCPI commands. For front panel key commands, refer to the *User's Guide* or to the Key help in the signal generator.

Loading a File from Non–Volatile Memory

Select the downloaded I/Q file in non–volatile waveform memory (NVWFM) and load it into volatile waveform memory (WFM1). The file comprises three items: I/Q data, marker file, and file header information.

Send one of the following SCPI command to copy the I/Q file, marker file and file header information:

```
:MEMory:COPY:NAME "<NVWFM:file_name>","<WFM1:file_name>"
:MEMory:COPY:NAME "<NVMKR:file_name>","<MKR1:file_name>"
:MEMory:COPY:NAME "<,"<HDR:file_name>"
```

NOTE When you copy a waveform file, marker file, or header file information from volatile or non–volatile memory, the waveform and associated marker and header files are all copied. Conversely, when you delete an I/Q file, the associated marker and header files are deleted. It is not necessary to send separate commands to copy or delete the marker and header files.

Playing the Waveform

NOTE If you would like to build and play a waveform *sequence*, refer to "Building and Playing Waveform Sequences" on page 241.

Play the waveform and use it to modulate the RF carrier.

1. List the waveform files from the volatile memory waveform list:

   Send the following SCPI command:

   ```
   :MMEMory:CATalog? "WFM1:"
   ```

2. Select the waveform from the volatile memory waveform list:

   Send the following SCPI command:

   ```
   :SOURce:RADio:ARB:WAVeform "WFM1:<file_name>"
   ```

3. Play the waveform:

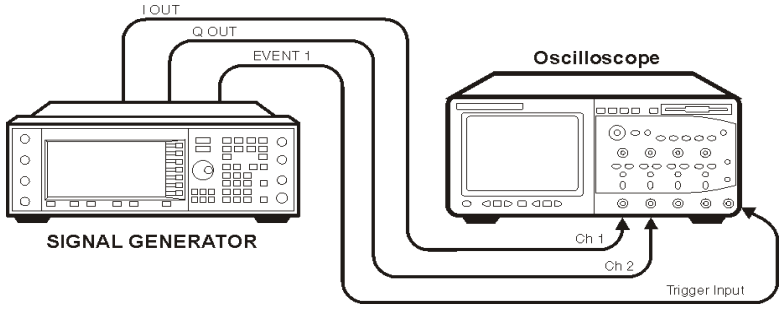
   Send the following SCPI commands:

   ```
   :SOURce:RADio:ARB:STATe ON
   :OUTPut:MODulation:STATe ON
   :OUTPut:STATe ON
   ```

Verifying the Waveform

Perform this procedure after completing the steps in the previous procedure, "Playing the Waveform" on page 240.

1. Connect the signal generator to an oscilloscope as shown in the figure.



2. Set an active marker point on the first waveform point for marker one.

NOTE Select the same waveform selected in "Playing the Waveform" on page 240.

Send the following SCPI commands:

```
:SOURce:RADio:ARB:MARKer:CLEar:ALL "WFM1:<file_name>",1
```

```
:SOURce:RADio:ARB:MARKer:SET "WFM1:<file_name>",1,1,1,0.
```

3. Compare the oscilloscope display to the plot of the I and Q data from the text file you created when you generated the data.

   If the oscilloscope display, and the I and Q data plots differ, recheck your code. For detailed information on programmatically creating and downloading waveform data, see "Creating Waveform Data" on page 227 and "Downloading Waveform Data" on page 233. For information on the waveform data requirements, see "Waveform Data Requirements" on page 199.

Building and Playing Waveform Sequences

The signal generator can be used to build waveform sequences. This section assumes you have created the waveform segment file(s) and have the waveform segment file(s) in volatile memory. The following SCPI commands can be used to generate and work with a waveform sequence. For more information refer to the signal generator's *SCPI Command Reference* and *User's Guide*.

NOTE If you would like to verify the waveform sequence, refer to "Verifying the Waveform" on page 241.

---

1. List the waveform files from the volatile memory waveform list:

   Send the following SCPI command:

   ```
   :MMEMory:CATalog? "WFM1:"
   ```

2. Select the waveform segment file(s) from the volatile memory waveform list:

   Send the following SCPI command:

   ```
   :SOURce:RADio:ARB:WAVeform "WFM1:<file_name>"
   ```

3. Save the waveform segment(s) ("<waveform1>", "<waveform2>", ...), to non–volatile memory as a waveform sequence ("<file_name>"), define the number of repetitions (<reps>), each waveform segment plays, and enable/disable markers (M1|M2|M3|M4|...), for each waveform segment:

   Send the following SCPI command:

   ```
   :SOURce:RADio:ARB:SEQuence
   "<file_name>","<waveform1>",<reps>,M1|M2|M3|M4,{"<waveform2>",<reps>,ALL}

   :SOURce:RADio:ARB:SEQuence? "<file_name>"
   ```

NOTE M1|M2|M3|M4 represent the number parameter of the marker selected (i.e. 1|2|3|4). Entering M1|M2|M3|M4 causes the signal generator to display an error. For more information on this SCPI command, refer to the signal generator's *SCPI Command Reference*.

---

4. Play the waveform sequence:

   Send the following SCPI commands:

   ```
   :SOURce:RADio:ARB:STATe ON
   :OUTPut:MODulation:STATe ON
   :OUTPut:STATe ON
   ```

Using the Download Utilities

Agilent provides free download utilities to download waveform data into the signal generator. The table in this section describes the capabilities of three such utilities.

For more information and to install the utilities, refer to the following URLs:

- Agilent Signal Studio Toolkit 2: *http://www.agilent.com/find/signalstudio*

  This software provides a graphical interface for downloading files.

- Agilent IntuiLink for Agilent PSG/ESG/E8663B Signal Generators: *http://www.agilent.com/find/intuilink*

  This software places icons in the Microsoft Excel and Word toolbar. Use the icons to connect to the signal generator and open a window for downloading files.

NOTE Agilent Intuilink is *not* available for the Agilent MXG.

- Agilent Waveform Download Assistant: *http://www.agilent.com/find/downloadassistant*

  This software provides functions for the MATLAB environment to download waveform data.

Features	Agilent Signal Studio Toolkit 2	Agilent IntuiLink[a]	Agilent Waveform Download Assistant
Downloads encrypted waveform files	X		
Downloads complex MATLAB waveform data			X
Downloads MATLAB files (.mat)	X		
Downloads unencrypted interleaved 16–bit I/Q files [b]	X	X	
Interleaves and downloads earlier 14–bit E443xB I and Q files[b]	X	X	
Swaps bytes for little endian order		X	
Manually select big endian byte order for 14–bit and 16–bit I/Q files	X		
Downloads user–created marker files	X	X	X
Performs scaling	X	X	X
Starts waveform play back	X		X
Sends SCPI Commands and Queries	X		X
Builds a waveform sequence	X		X

a. Agilent Intuilink is *not* available for the Agilent MXG.
b. ASCII or binary format.

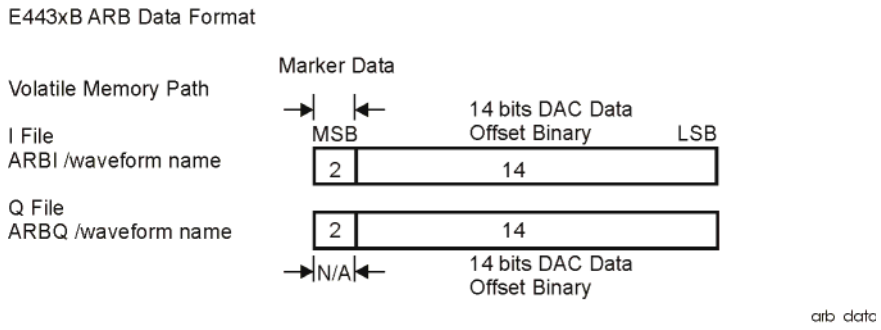
Downloading E443xB Signal Generator Files

To download earlier E443xB model I and Q files, use the same SCPI commands as if downloading files to an E443xB signal generator. The signal generator automatically converts the E443xB files to the proper file format as described in "Waveform Structure" on page 207 and stores them in the signal generator's memory. This conversion process causes the signal generator to take more time to download the earlier file format. To minimize the time to convert earlier E443xB files to the proper file format, store E443xB file downloads to volatile memory, and then transfer them over to non–volatile (NVWFM) memory.

NOTE You cannot extract waveform data downloaded as E443xB files.

E443xB Data Format

The following diagram describes the data format for the E443xB waveform files. This file structure can be compared with the new style file format shown in "Waveform Structure" on page 207. If you create new waveform files for the signal generator, use the format shown in "Waveform Data Requirements" on page 199.



Storage Locations for E443xB ARB files

Place waveforms in either volatile memory or non–volatile memory. The signal generator supports the E443xB directory structure for waveform file downloads (i.e. "ARBI:", "ARBQ:", "NVARBI:", and "NVARBQ:", see also "SCPI Commands" on page 246).

Volatile Memory Storage Locations

- /user/arbi/
- /user/arbq/

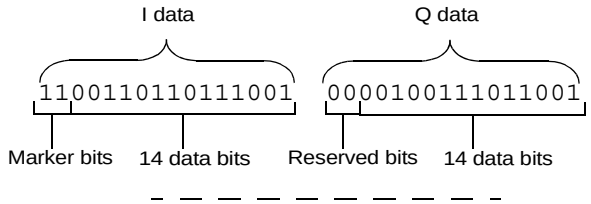
Non–Volatile Memory Storage Locations

- /user/nvarbi/
- /user/nvarbq/

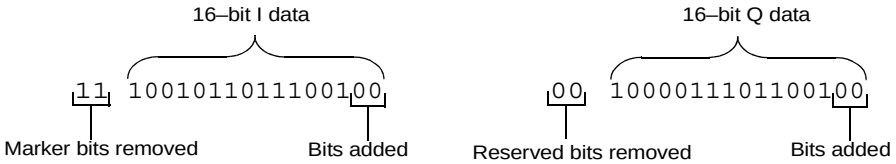
Loading files into the above directories (volatile or non–volatile memory) does not actually store them in those directories. Instead, these directories function as "pipes" to the format translator. The signal generator performs the following functions on the E443xB data:

- Converts the 14–bit I and Q data into 16–bit data (the format required by the signal generator). Subtract 8192, left shifts the data, and appends two bits (zeros) before the least significant bit (i.e. the offset binary values are converted to 2's complement values by the signal generator).

E443xB 14–Bit Data

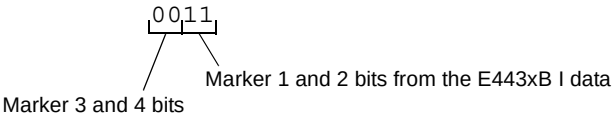


Subtracts 8192, Left Shifts, and Adds Zeros—Removes Marker and Reserved Bits (16–Bit Data Format)



- Creates a marker file and places the marker information, bits 14 and 15 of the E443xB I data, into the marker file for markers one and two. Markers three and four, within the new marker file, are set to zero (off).

Places the I Marker Bits into the Signal Generator Marker File



- Interleaves the 16–bit I and Q data creating one I/Q file.
- Creates a file header with all parameters set to unspecified (factory default file header setting).

SCPI Commands

Use the following commands to download E443xB waveform files into the signal generator.

NOTE To avoid overwriting the current waveform in volatile memory, before downloading files into volatile memory (`WFM1`), change the file name or turn off the ARB. For more information, on manually turning off the ARB, refer to the *User's Guide*.

To turn off the ARB remotely, send: `:SOURce:RADio:ARB:STATe OFF`.

Extraction Method/ Memory Type	Command Syntax Options
SCPI/ volatile memory	`:MMEM:DATA "ARBI:<file_name>", <I waveform block data>` `:MMEM:DATA "ARBQ:<file_name>", <Q waveform data>`
SCPI/ non–volatile memory	`:MMEM:DATA "NVARBI:<file_name>", <I waveform block data>` `:MMEM:DATA "NVARBQ:<file_name>", <Q waveform block data>`

The variables `<I waveform block data>` and `<Q waveform block data>` represents data in the E443xB file format. The string variable `<file_name>` is the name of the I and Q data file. After downloading the data, the signal generator associates a file header and marker file with the I/Q data file.

Programming Examples

NOTE The programming examples contain instrument–specific information. However, users can still use these programming examples by substituting in the instrument–specific information for your signal generator. Model specific exceptions for programming use, will be noted at the top of each programming section.

The programming examples use GPIB or LAN interfaces and are written in the following languages:

- C++ (page 247)
- MATLAB (page 268)
- Visual Basic (page 275)
- HP Basic (page 280)

See Chapter 2 of this programming guide for information on interfaces and IO libraries.

The example programs are also available on the signal generator Documentation CD–ROM, which allows you to cut and paste the examples into an editor.

C++ Programming Examples

This section contains the following programming examples:

- "Creating and Storing Offset I/Q Data–Big and Little Endian Order" on page 247
- "Creating and Storing I/Q Data–Little Endian Order" on page 251
- "Creating and Downloading I/Q Data–Big and Little Endian Order" on page 252
- "Importing and Downloading I/Q Data–Big Endian Order" on page 256
- "Importing and Downloading Using VISA–Big Endian Order" on page 258
- "Importing, Byte Swapping, Interleaving, and Downloading I and Q Data–Big and Little Endian Order" on page 262

Creating and Storing Offset I/Q Data—Big and Little Endian Order

On the documentation CD, this programming example's name is "*offset_iq_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) follows the same coding algorithm as the MATLAB programming example "Creating and Storing I/Q Data" on page 268 and performs the following functions:

- error checking
- data creation
- data normalization
- data scaling
- I/Q signal offset from the carrier (single sideband suppressed carrier signal)
- byte swapping and interleaving for little endian order data
- I and Q interleaving for big endian order data
- binary data file storing to a PC or workstation
- reversal of the data formatting process (byte swapping, interleaving, and normalizing the data)

After creating the binary file, you can use FTP, one of the download utilities, or one of the C++ download programming examples to download the file to the signal generator.

```
// This C++ example shows how to
// 1.) Create a simple IQ waveform
// 2.) Save the waveform into the ESG/PSG Internal Arb format
//       This format is for the E4438C, E8267C, E8267D
//       This format will not work with the ESG E443xB or the Agilent MXG N518xA
// 3.) Load the internal Arb format file into an array

#include <stdio.h>
#include <string.h>
#include <math.h>

const int POINTS = 1000;  // Size of waveform
const char *computer = "PCWIN";

int main(int argc, char* argv[])
{
```

```
// 1.) Create Simple IQ Signal ***************************************
 // This signal is a single tone on the upper
 // side of the carrier and is usually refered to as
 // a Single Side Band Suppressed Carrier (SSBSC) signal.
 // It is nothing more than a cosine wavefomm in I
 // and a sine waveform in Q.

 int points = POINTS; // Number of points in the waveform
 int cycles = 101; // Determines the frequency offset from the carrier
 double Iwave[POINTS]; // I waveform
 double Qwave[POINTS]; // Q waveform
 short int waveform[2*POINTS]; // Holds interleaved I/Q data
 double maxAmp = 0; // Used to Normalize waveform data
 double minAmp = 0; // Used to Normalize waveform data
 double scale = 1;
 char buf; // Used for byte swapping
 char *pChar; // Used for byte swapping
 bool PC = true; // Set flag as appropriate

 double phaseInc = 2.0 * 3.141592654 * cycles / points;
 double phase = 0;
 int i = 0;
 for( i=0; i<points; i++ )
 {
   phase = i * phaseInc;
   Iwave[i] = cos(phase);
   Qwave[i] = sin(phase);
 }

 // 2.) Save waveform in internal format *********************************
 // Convert the I and Q data into the internal arb format
 // The internal arb format is a single waveform containing interleaved IQ
 // data. The I/Q data is signed short integers (16 bits).
 // The data has values scaled between +-32767 where
 //   DAC Value   Description
 //    32767      Maximum positive value of the DAC
 //        0      Zero out of the DAC
 //   -32767      Maximum negative value of the DAC
 // The internal arb expects the data bytes to be in Big Endian format.
 // This is opposite of how short integers are saved on a PC (Little Endian).
 // For this reason the data bytes are swapped before being saved.
```

```
// Find the Maximum amplitude in I and Q to normalize the data between +-1
maxAmp = Iwave[0];
minAmp = Iwave[0];
for( i=0; i<points; i++)
{
  if( maxAmp < Iwave[i] )
      maxAmp = Iwave[i];
   else if( minAmp > Iwave[i] )
      minAmp = Iwave[i];

  if( maxAmp < Qwave[i] )
      maxAmp = Qwave[i];
  else if( minAmp > Qwave[i] )
    minAmp = Qwave[i];
}
maxAmp = fabs(maxAmp);
minAmp = fabs(minAmp);
if( minAmp > maxAmp )
  maxAmp = minAmp;

// Convert to short integers and interleave I/Q data
scale = 32767 / maxAmp;     // Watch out for divide by zero.
for( i=0; i<points; i++)
{
  waveform[2*i] = (short)floor(Iwave[i]*scale + 0.5);
  waveform[2*i+1] = (short)floor(Qwave[i]*scale + 0.5);
}
// If on a PC swap the bytes to Big Endian
if( strcmp(computer,"PCWIN") == 0 )
//if( PC )
{
  pChar = (char *)&waveform[0];   // Character pointer to short int data
  for( i=0; i<2*points; i++ )
  {
     buf = *pChar;
     *pChar = *(pChar+1);
     *(pChar+1) = buf;
     pChar+= 2;
   }
}
// Save the data to a file
// Use FTP or one of the download assistants to download the file to the
```

```
  // signal generator
  char *filename = "C:\\Temp\\PSGTestFile";
  FILE *stream = NULL;
  stream = fopen(filename, "w+b");// Open the file
  if (stream==NULL) perror ("Cannot Open File");
  int numwritten = fwrite( (void *)waveform, sizeof( short ), points*2, stream );
  fclose(stream);// Close the file

  // 3.) Load the internal Arb format file ********************************
  // This process is just the reverse of saving the waveform
  // Read in waveform as unsigned short integers.
  // Swap the bytes as necessary
  // Normalize between +-1
  // De-interleave the I/Q Data
  // Open the file and load the internal format data
  stream = fopen(filename, "r+b");// Open the file
  if (stream==NULL) perror ("Cannot Open File");
  int numread = fread( (void *)waveform, sizeof( short ),  points*2, stream );
  fclose(stream);// Close the file
  // If on a PC swap the bytes back to Little Endian
  if( strcmp(computer,"PCWIN") == 0 )
  {
     pChar = (char *)&waveform[0];   // Character pointer to short int data
     for( i=0; i<2*points; i++ )
     {
        buf = *pChar;
        *pChar = *(pChar+1);
        *(pChar+1) = buf;
        pChar+= 2;
     }
  }
  // Normalize De-Interleave the IQ data
  double IwaveIn[POINTS];
  double QwaveIn[POINTS];
  for( i=0; i<points; i++)
  {
    IwaveIn[i] = waveform[2*i] / 32767.0;
    QwaveIn[i] = waveform[2*i+1] / 32767.0;
  }
  return 0;
}
```

Creating and Storing I/Q Data—Little Endian Order

On the documentation CD, this programming example's name is "*CreateStore_Data_c++.txt*."

This C++ programming example (compiled using Metrowerks CodeWarrior 3.0) performs the following functions:

- error checking
- data creation
- byte swapping and interleaving for little endian order data
- binary data file storing to a PC or workstation

After creating the binary file, you can use FTP, one of the download utilities, or one of the C++ download programming examples to download the file to the signal generator.

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <stdlib.h>

using namespace std;

int main ( void )
{
    ofstream out_stream;       // write the I/Q data to a file
    const unsigned int SAMPLES =200;    // number of sample pairs in the waveform
    const short AMPLITUDE = 32000;      // amplitude between 0 and full scale dac value
    const double two_pi = 6.2831853;

    //allocate buffer for waveform
    short* iqData = new short[2*SAMPLES];// need two bytes for each integer
    if (!iqData)
    {
       cout << "Could not allocate data buffer." << endl;
       return 1;
    }

    out_stream.open("IQ_data");// create a data file
    if (out_stream.fail())
    {
      cout << "Input file opening failed" << endl;
      exit(1);
    }
    //generate the sample data for I and Q. The I channel will have a sine
    //wave and the Q channel will a cosine wave.
```

```
    for (int i=0; i<SAMPLES; ++i)
    {
        iqData[2*i] = AMPLITUDE * sin(two_pi*i/(float)SAMPLES);
        iqData[2*i+1] = AMPLITUDE * cos(two_pi*i/(float)SAMPLES);
    }
   // make sure bytes are in the order MSB(most significant byte) first. (PC only).

    char* cptr = (char*)iqData;// cast the integer values to characters

    for (int i=0; i<(4*SAMPLES); i+=2)// 4*SAMPLES
    {
        char temp = cptr[i];// swap LSB and MSB bytes
        cptr[i]=cptr[i+1];
        cptr[i+1]=temp;
    }

    // now write the buffer to a file

        out_stream.write((char*)iqData, 4*SAMPLES);
 return 0;
}
```

Creating and Downloading I/Q Data—Big and Little Endian Order

On the documentation CD, this programming example's name is "*CreateDwnLd_Data_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) performs the following functions:

- error checking
- data creation
- data scaling
- text file creation for viewing and debugging data
- byte swapping and interleaving for little endian order data
- interleaving for big endian order data
- data saving to an array (data block)
- data block download to the signal generator

```
// This C++ program is an example of creating and scaling
// I and Q data, and then downloading the data into the
// signal generator as an interleaved I/Q file.
// This example uses a sine and cosine wave as the I/Q
// data.
//
// Include the standard headers for SICL programming
#include <sicl.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// Choose a GPIB, LAN, or RS-232 connection
char* instOpenString =”lan[galqaDhcp1]”;
//char* instOpenString =”gpib0,19”;

// Pick some maximum number of samples, based on the
// amount of memory in your computer and the signal generator.
const int NUMSAMPLES=500;

int main(int argc, char* argv[])
{
     // Create a text file to view the waveform
     // prior to downloading it to the signal generator.
     // This verifies that the data looks correct.

    char *ofile = “c:\\temp\\iq.txt”;

    // Create arrays to hold the I and Q data

    int idata[NUMSAMPLES];
    int qdata[NUMSAMPLES];

    // save the number of sampes into numsamples
    int numsamples = NUMSAMPLES;

    // Fill the I and Q buffers with the sample data
    for(int index=0; index<numsamples; index++)
    {
         // Create the I and Q data for the number of waveform
        // points and Scale the data (20000 * ...) as a precentage
        // of the DAC full scale (-32768 to 32767). This example
        // scales to approximately 70% of full scale.
        idata[index]=23000 * sin((4*3.14*index)/numsamples);
        qdata[index]=23000 * cos((4*3.14*index)/numsamples);
    }

    // Print the I and Q values to a text file. View the data
    // to see if its correct and if needed, plot the data in a
```

```
// spreadsheet to help spot any problems.
FILE *outfile = fopen(ofile, "w");

if (outfile==NULL) perror ("Error opening file to write");
for(index=0; index<numsamples; index++)
{
    fprintf(outfile, "%d, %d\n", idata[index], qdata[index]);
}
fclose(outfile);

// Little endian order data, use the character array and for loop.
// If big endian order, comment out this character array and for loop,
// and use the next loop (Big Endian order data).

// We need a buffer to interleave the I and Q data.
// 4 bytes to account for 2 I bytes and 2 Q bytes.

char iqbuffer[NUMSAMPLES*4];

// Interleave I and Q, and swap bytes from little
// endian order to big endian order.
for(index=0; index<numsamples; index++)
{
    int ivalue = idata[index];
    int qvalue = qdata[index];
    iqbuffer[index*4]   = (ivalue >> 8) & 0xFF; // high byte of i
    iqbuffer[index*4+1] = ivalue & 0xFF;        // low byte of i
    iqbuffer[index*4+2] = (qvalue >> 8) & 0xFF; // high byte of q
    iqbuffer[index*4+3] = qvalue & 0xFF;        // low byte of q
}

// Big Endian order data, uncomment the following lines of code.
// Interleave the I and Q data.

// short iqbuffer[NUMSAMPLES*2];            // Big endian order, uncomment this line
// for(index=0; index<numsamples; index++)  // Big endian order, uncomment this line
// {                                        // Big endian order, uncomment this line
//     iqbuffer[index*2]   = idata[index];  // Big endian order, uncomment this line
//     iqbuffer[index*2+1] = qdata[index];  // Big endian order, uncomment this line
// }                                        // Big endian order, uncomment this line

// Open a connection to write to the instrument
INST id=iopen(instOpenString);
if (!id)
```

```
    {
        fprintf(stderr, "iopen failed (%s)\n", instOpenString);
        return -1;
    }

    // Declare variables to hold portions of the SCPI command
    int bytesToSend;
    char s[20];
    char cmd[200];

    bytesToSend = numsamples*4;       // calculate the number of bytes
    sprintf(s, "%d", bytesToSend); // create a string s with that number of bytes

    // The SCPI command has four parts.
    //   Part 1 = :MEM:DATA "filename",#
    //   Part 2 = length of Part 3 when written to a string
    //   Part 3 = length of the data in bytes.  This is in s from above.
    //   Part 4 = the buffer of data

    // Build parts 1, 2, and 3 for the I and Q data.
    sprintf(cmd, ":MEM:DATA \"WFM1:FILE1\", #%d%d", strlen(s), bytesToSend);
    // Send parts 1, 2, and 3
    iwrite(id, cmd, strlen(cmd), 0, 0);
    // Send part 4.  Be careful to use the correct command here.  In many
    // programming languages, there are two methods to send SCPI commands:
    //   Method 1 = stop at the first '0' in the data
    //   Method 2 = send a fixed number of bytes, ignoring '0' in the data.
    // You must find and use the correct command for Method 2.
    iwrite(id, iqbuffer, bytesToSend, 0, 0);
    // Send a terminating carriage return
    iwrite(id, "\n", 1, 1, 0);

    printf("Loaded file using the E4438C, E8267C and E8267D format\n");
    return 0;
}
```

Importing and Downloading I/Q Data—Big Endian Order

On the documentation CD, this programming example's name is "*impDwnLd_c++.txt*."

This C++ programming example (compiled using Metrowerks CodeWarrier 3.0) assumes that the data is in big endian order and performs the following functions:

- error checking
- binary file importing from the PC or workstation.
- binary file download to the signal generator.

```
// Description: Send a file in blocks of data to a signal generator
//
#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


// ATTENTION:
// - Configure these three lines appropriately for your instrument
//   and use before compiling and running
//
char* instOpenString = "gpib7,19"; //for LAN replace with "lan[<hostname or IP address>]"
const char* localSrcFile = "D:\\home\\TEST_WAVE"; //enter file location on PC/workstation
const char* instDestFile = "/USER/BBG1/WAVEFORM/TEST_WAVE"; //for non-volatile memory
                                                            //remove BBG1 from file path
// Size of the copy buffer
const int BUFFER_SIZE = 100*1024;


int
main()
{
    INST id=iopen(instOpenString);
    if (!id)
    {
        fprintf(stderr, "iopen failed (%s)\n", instOpenString);
        return -1;
    }

    FILE* file = fopen(localSrcFile, "rb");
    if (!file)
    {
        fprintf(stderr, "Could not open file: %s\n", localSrcFile);
        return 0;
    }
```

```
if( fseek( file, 0, SEEK_END ) < 0 )
{
    fprintf(stderr,"Cannot seek to the end of file.\n" );
    return 0;
}

long lenToSend = ftell(file);
printf("File size = %d\n", lenToSend);

if (fseek(file, 0, SEEK_SET) < 0)
{
    fprintf(stderr,"Cannot seek to the start of file.\n");
    return 0;
}

char* buf = new char[BUFFER_SIZE];
if (buf && lenToSend)
{
    // Prepare and send the SCPI command header
    char s[20];
    sprintf(s, "%d", lenToSend);
    int lenLen = strlen(s);
    char s2[256];
    sprintf(s2, "mmem:data \"%s\", #%d%d", instDestFile, lenLen, lenToSend);
    iwrite(id, s2, strlen(s2), 0, 0);

    // Send file in BUFFER_SIZE chunks
    long numRead;
    do
    {
        numRead = fread(buf, sizeof(char), BUFFER_SIZE, file);
        iwrite(id, buf, numRead, 0, 0);
    } while (numRead == BUFFER_SIZE);

    // Send the terminating newline and EOM
    iwrite(id, "\n", 1, 1, 0);

    delete [] buf;
}
else
{
```

```
        fprintf(stderr, "Could not allocate memory for copy buffer\n");
    }

    fclose(file);
    iclose(id);
    return 0;
}
```

Importing and Downloading Using VISA—Big Endian Order

On the documentation CD, this programming example's name is "*DownLoad_Visa_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) assumes that the data is in big endian order and performs the following functions:

- error checking
- binary file importing from the PC or workstation
- binary file download to the signal generator's non–volatile memory

To load the waveform data to volatile (WFM1) memory, change the instDestfile declaration to: "USER/BBG1/WAVEFORM/".

```
//**************************************************************************************
// PROGRAM NAME:Download_Visa_c++.cpp
//
// PROGRAM DESCRIPTION:Sample test program to download ARB waveform data. Send a
// file in chunks of ascii data to the signal generator.
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the LAN/TCPIP to download a file to the signal generator's
// non-volatile memory. The program allocates a memory buffer on the PC or
// workstation of 102400 bytes (100*1024 bytes). The actual size of the buffer is
// limited by the memory on your PC or workstation, so the buffer size can be
// increased or decreased to meet your system limitations.
//
// While this program uses the LAN/TCPIP to download a waveform file into
// non-volatile memory, it can be modified to store files in volatile memory
// WFM1 using GPIB by setting the instrOpenString = "TCPIP0::xxx.xxx.xxx.xxx::INSTR"
// declaration with "GPIB::19::INSTR"
//
// The program also includes some error checking to alert you when problems arise
// while trying to download files. This includes checking to see if the file exists.
//**************************************************************************************
// IMPORTANT: Replace the xxx.xxx.xxx.xxx IP address in the instOpenString declaration
// in the code below with the IP address of your signal generator. (or you can use the
// instrument's hostname). Replace the localSrcFile and instDestFile directory paths
```

```
// as needed.
//*****************************************************************************


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include  "visa.h"
//
// IMPORTANT:
//   Configure the following three lines correctly before compiling and running

char* instOpenString ="TCPIP0::xxx.xxx.xxx.xxx::INSTR"; // your instrument's IP address

const char* localSrcFile = "\\Files\\IQ_DataC";

const char* instDestFile = "/USER/WAVEFORM/IQ_DataC";

const int BUFFER_SIZE = 100*1024;// Size of the copy buffer

int main(int argc, char* argv[])
{
    ViSession defaultRM, vi;
    ViStatus status = 0;

    status = viOpenDefaultRM(&defaultRM);// Open the default resource manager

    // TO DO: Error handling here

    status = viOpen(defaultRM, instOpenString, VI_NULL, VI_NULL, &vi);

    if (status)// If any errors then display the error and exit the program
    {
        fprintf(stderr, "viOpen failed (%s)\n", instOpenString);
return -1;
    }

    FILE* file = fopen(localSrcFile, "rb");// Open local source file for binary reading

    if (!file) // If any errors display the error and exit the program
    {
        fprintf(stderr, "Could not open file: %s\n", localSrcFile);
```

```
return 0;
    }

    if( fseek( file, 0, SEEK_END ) < 0 )
    {
        fprintf(stderr,"Cannot lseek to the end of file.\n" );
        return 0;
    }

    long lenToSend = ftell(file);// Number of bytes in the file

    printf("File size = %d\n", lenToSend);

    if (fseek(file, 0, SEEK_SET) < 0)
    {
        fprintf(stderr,"Cannot lseek to the start of file.\n");
        return 0;
    }

    unsigned char* buf = new unsigned char[BUFFER_SIZE]; // Allocate char buffer memory

    if (buf && lenToSend)
    {
        // Do not send the EOI (end of instruction) terminator on any write except the
        // last one

        viSetAttribute( vi, VI_ATTR_SEND_END_EN, 0 );

        // Prepare and send the SCPI command header

        char s[20];
        sprintf(s, "%d", lenToSend);

        int lenLen = strlen(s);
        unsigned char s2[256];

// Write the command mmem:data and the header.The number lenLen represents the
// number of bytes and the actual number of bytes is the variable lenToSend

        sprintf((char*)s2, "mmem:data \"%s\", #%d%d", instDestFile, lenLen, lenToSend);

// Send the command and header to the signal generator
```

```
        viWrite(vi, s2, strlen((char*)s2), 0);

        long numRead;

// Send file in BUFFER_SIZE chunks to the signal generator

        do
        {
            numRead = fread(buf, sizeof(char), BUFFER_SIZE, file);

            viWrite(vi, buf, numRead, 0);

        } while (numRead == BUFFER_SIZE);

        // Send the terminating newline and EOI

        viSetAttribute( vi, VI_ATTR_SEND_END_EN, 1 );

        char* newLine = "\n";

        viWrite(vi, (unsigned char*)newLine, 1, 0);

        delete [] buf;
    }
    else
    {
        fprintf(stderr, "Could not allocate memory for copy buffer\n");
    }

    fclose(file);
    viClose(vi);
    viClose(defaultRM);

    return 0;
}
```

Importing, Byte Swapping, Interleaving, and Downloading I and Q Data—Big and Little Endian Order

On the documentation CD, this programming example's name is "*impDwnLd2_c++.txt*."

This C++ programming example (compiled using Microsoft Visual C++ 6.0) performs the following functions:

- error checking
- binary file importing (earlier E443xB or current model signal generators)
- byte swapping and interleaving for little endian order data
- data interleaving for big endian order data
- data scaling
- binary file download for earlier E443xB data or current signal generator formatted data

```
// This C++ program is an example of loading I and Q
// data into an E443xB, E4438C, E8267C, or E8267D signal
// generator.
//
// It reads the I and Q data from a binary data file
// and then writes the data to the instrument.

// Include the standard headers for SICL programming
#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Choose a GPIB, LAN, or RS-232 connection
char* instOpenString ="gpib0,19";

// Pick some maximum number of samples, based on the
// amount of memory in your computer and your waveforms.
const int MAXSAMPLES=50000;

int main(int argc, char* argv[])

{
    // These are the I and Q input files.
    // Some compilers will allow '/' in the directory
    // names.  Older compilers might need '\\' in the
    // directory names.  It depends on your operating system
    // and compiler.
    char *ifile = "c:\\SignalGenerator\\data\\BurstA1I.bin";
    char *qfile = "c:\\SignalGenerator\\data\\BurstA1Q.bin";
```

```
// This is a text file to which we will write the
// I and Q data just for debugging purposes.  It is
// a good programming practice to check your data
// in this way before attempting to write it to
// the instrument.
char *ofile = "c:\\SignalGenerator\\data\\iq.txt";

// Create arrays to hold the I and Q data
int idata[MAXSAMPLES];
int qdata[MAXSAMPLES];

// Often we must modify, scale, or offset the data
// before loading it into the instrument.  These
// buffers are used for that purpose.  Since each
// sample is 16 bits, and a character only holds
// 8 bits, we must make these arrays twice as long
// as the I and Q data arrays.
char ibuffer[MAXSAMPLES*2];
char qbuffer[MAXSAMPLES*2];

// For the E4438C or E8267C/67D, we might also need to interleave
// the I and Q data.  This buffer is used for that
// purpose.  In this case, this buffer must hold
// both I and Q data so it needs to be four times
// as big as the data arrays.
char iqbuffer[MAXSAMPLES*4];

// Declare variables which will be used later
bool done;
FILE *infile;
int index, numsamples, i1, i2, ivalue;

// In this example, we'll assume the data files have
// the I and Q data in binary form as unsigned 16 bit integers.
// This next block reads those binary files.  If your
// data is in some other format, then replace this block
// with appropriate code for reading your format.
// First read I values
done = false;
index = 0;
infile = fopen(ifile, "rb");
if (infile==NULL) perror ("Error opening file to read");
```

```
while(!done)
{
    i1 = fgetc(infile);  // read the first byte
    if(i1==EOF) break;
    i2 = fgetc(infile);  // read the next byte
    if(i2==EOF) break;
    ivalue=i1+i2*256;    // put the two bytes together
    // note that the above format is for a little endian
    // processor such as Intel.  Reverse the order for
    // a big endian processor such as Motorola, HP, or Sun
    idata[index++]=ivalue;
    if(index==MAXSAMPLES) break;
}
fclose(infile);

// Then read Q values
index = 0;
infile = fopen(qfile, "rb");
if (infile==NULL) perror ("Error opening file to read");
while(!done)
{
    i1 = fgetc(infile);  // read the first byte
    if(i1==EOF) break;
    i2 = fgetc(infile);  // read the next byte
    if(i2==EOF) break;
    ivalue=i1+i2*256;    // put the two bytes together
    // note that the above format is for a little endian
    // processor such as Intel.  Reverse the order for
    // a big endian processor such as Motorola, HP, or Sun
    qdata[index++]=ivalue;
    if(index==MAXSAMPLES) break;
}
fclose(infile);

// Remember the number of samples which were read from the file.
numsamples = index;

// Print the I and Q values to a text file.  If you are
// having trouble, look in the file and see if your I and
// Q data looks correct.  Plot the data from this file if
// that helps you to diagnose the problem.
FILE *outfile = fopen(ofile, "w");
```

```
if (outfile==NULL) perror ("Error opening file to write");
for(index=0; index<numsamples; index++)
{
    fprintf(outfile, "%d, %d\n", idata[index], qdata[index]);
}
fclose(outfile);

// The E443xB, E4438C, E8267C or E8267D all use big-endian
// processors.  If your software is running on a little-endian
// processor such as Intel, then you will need to swap the
// bytes in the data before sending it to the signal generator.

// The arrays ibuffer and qbuffer are used to hold the data
// after any byte swapping, shifting or scaling.

// In this example, we'll assume that the data is in the format
// of the E443xB without markers.  In other words, the data
// is in the range 0-16383.
//  0 gives negative full-scale output
//  8192 gives 0 V output
//  16383 gives positive full-scale output
// If this is not the scaling of your data, then you will need
// to scale your data appropriately in the next two blocks.

// ibuffer and qbuffer will hold the data in the E443xB format.
// No scaling is needed, however we need to swap the byte order
// on a little endian computer.  Remove the byte swapping
// if you are using a big endian computer.
for(index=0; index<numsamples; index++)
{
    int ivalue = idata[index];
    int qvalue = qdata[index];
    ibuffer[index*2]   = (ivalue >> 8) & 0xFF; // high byte of i
    ibuffer[index*2+1] = ivalue & 0xFF;        // low byte of i
    qbuffer[index*2]   = (qvalue >> 8) & 0xFF; // high byte of q
    qbuffer[index*2+1] = qvalue & 0xFF;        // low byte of q
}

// iqbuffer will hold the data in the E4438C, E8267C, E8267D
// format.  In this format, the I and Q data is interleaved.
// The data is in the range -32768 to 32767.
//   -32768 gives negative full-scale output
```

```
//        0 gives 0 V output
//    32767 gives positive full-scale output
// From these ranges, it appears you should offset the
// data by 8192 and scale it by 4.  However, due to the
// interpolators in these products, it is better to scale
// the data by a number less than four.  Commonly a good
// choice is 70% of 4 which is 2.8.
// By default, the signal generator scales data to 70%
// If you scale the data here, you may want to change the
// signal generator scaling to 100%
// Also we need to swap the byte order on a little endian
// computer.  This code also works for big endian order data
// since it swaps bytes based on the order.
for(index=0; index<numsamples; index++)
{
    int iscaled = 2.8*(idata[index]-8192); // shift and scale
    int qscaled = 2.8*(qdata[index]-8192); // shift and scale
    iqbuffer[index*4]   = (iscaled >> 8) & 0xFF; // high byte of i
    iqbuffer[index*4+1] = iscaled & 0xFF;        // low byte of i
    iqbuffer[index*4+2] = (qscaled >> 8) & 0xFF; // high byte of q
    iqbuffer[index*4+3] = qscaled & 0xFF;  // low byte of q
}

// Open a connection to write to the instrument
INST id=iopen(instOpenString);
if (!id)
{
    fprintf(stderr, "iopen failed (%s)\n", instOpenString);
    return -1;
}

// Declare variables which will be used later
int bytesToSend;
char s[20];
char cmd[200];

// The E4438C, E8267C and E8267D accept the E443xB format.
// so we can use this next section on any of these signal generators.
// However the E443xB format only uses 14 bits.

bytesToSend = numsamples*2;       // calculate the number of bytes
sprintf(s, "%d", bytesToSend); // create a string s with that number of bytes
```

```
// The SCPI command has four parts.
//   Part 1 = :MEM:DATA "filename",
//   Part 2 = length of Part 3 when written to a string
//   Part 3 = length of the data in bytes.  This is in s from above.
//   Part 4 = the buffer of data

// Build parts 1, 2, and 3 for the I data.
sprintf(cmd, ":MEM:DATA \"ARBI:FILE1\", #%d%d", strlen(s), bytesToSend);
// Send parts 1, 2, and 3
iwrite(id, cmd, strlen(cmd), 0, 0);
// Send part 4.  Be careful to use the correct command here.  In many
// programming languages, there are two methods to send SCPI commands:
//   Method 1 = stop at the first '0' in the data
//   Method 2 = send a fixed number of bytes, ignoring '0' in the data.
// You must find and use the correct command for Method 2.
iwrite(id, ibuffer, bytesToSend, 0, 0);
// Send a terminating carriage return
iwrite(id, "\n", 1, 1, 0);

// Identical to the section above, except for the Q data.
sprintf(cmd, ":MEM:DATA \"ARBQ:FILE1\", #%d%d", strlen(s),bytesToSend);
iwrite(id, cmd, strlen(cmd), 0, 0);
iwrite(id, qbuffer, bytesToSend, 0, 0);
iwrite(id, "\n", 1, 1, 0);

printf("Loaded FILE1 using the E443xB format\n");

// The E4438C, E8267C and E8267D have a newer faster format which
// allows 16 bits to be used.  However this format is not accepted in
// the E443xB.  Therefore do not use this next section for the E443xB.

printf("Note: Loading FILE2 on a E443xB will cause \"ERROR: 208, I/O error\"\n");

// Identical to the I and Q sections above except
//   a) The I and Q data are interleaved
//   b) The buffer of I+Q is twice as long as the I buffer was.
//   c) The SCPI command uses WFM1 instead of ARBI and ARBQ.
bytesToSend = numsamples*4;
sprintf(s, "%d", bytesToSend);
sprintf(cmd, ":mem:data \"WFM1:FILE2\", #%d%d", strlen(s),bytesToSend);
iwrite(id, cmd, strlen(cmd), 0, 0);
```

```
    iwrite(id, iqbuffer, bytesToSend, 0, 0);
    iwrite(id, "\n", 1, 1, 0);
    printf("Loaded FILE2 using the E4438C, E8267C and E8267D format\n");
    return 0;
}
```

MATLAB Programming Examples

This section contains the following programming examples:

- "Creating and Storing I/Q Data" on page 268
- "Creating and Downloading a Pulse" on page 271

Creating and Storing I/Q Data

On the documentation CD, this programming example's name is "*offset_iq_ml.m*."

This MATLAB programming example follows the same coding algorithm as the C++ programming example "Creating and Storing Offset I/Q Data–Big and Little Endian Order" on page 247 and performs the following functions:

- error checking
- data creation
- data normalization
- data scaling
- I/Q signal offset from the carrier (single sideband suppressed carrier signal)
- byte swapping and interleaving for little endian order data
- I and Q interleaving for big endian order data
- binary data file storing to a PC or workstation
- reversal of the data formatting process (byte swapping, interleaving, and normalizing the data)

```
function main
% Using MatLab this example shows how to
% 1.) Create a simple IQ waveform
% 2.) Save the waveform into the Agilent MXG/ESG/PSG Internal Arb format
%     This format is for the N5162A/82A, E4438C, E8267C, and E8267D
%     This format will not work with the earlier E443xB ESG
% 3.) Load the internal Arb format file into a MatLab array

% 1.) Create Simple IQ Signal ***************************************
% This signal is a single tone on the upper
% side of the carrier and is usually refered to as
% a Single Side Band Suppressed Carrier (SSBSC) signal.
% It is nothing more than a cosine wavefomm in I
% and a sine waveform in Q.
%
points = 1000;      % Number of points in the waveform
cycles = 101;       % Determines the frequency offset from the carrier
```

```
phaseInc = 2*pi*cycles/points;
phase = phaseInc * [0:points-1];

Iwave = cos(phase);
Qwave = sin(phase);

% 2.) Save waveform in internal format *********************************
% Convert the I and Q data into the internal arb format
% The internal arb format is a single waveform containing interleaved IQ
% data. The I/Q data is signed short integers (16 bits).
% The data has values scaled between +-32767 where
%   DAC Value   Description
%    32767      Maximum positive value of the DAC
%        0      Zero out of the DAC
%   -32767      Maximum negative value of the DAC
% The internal arb expects the data bytes to be in Big Endian format.
% This is opposite of how short integers are saved on a PC (Little Endian).
% For this reason the data bytes are swapped before being saved.

% Interleave the IQ data
waveform(1:2:2*points) = Iwave;
waveform(2:2:2*points) = Qwave;
%[Iwave;Qwave];
%waveform = waveform(:)';

% Normalize the data between +-1
waveform = waveform / max(abs(waveform));   % Watch out for divide by zero.

% Scale to use full range of the DAC
waveform = round(waveform * 32767);     % Data is now effectively signed short integer values

% waveform = round(waveform * (32767 / max(abs(waveform))));    % More efficient than previous two
steps!

% PRESERVE THE BIT PATTERN but convert the waveform to
% unsigned short integers so the bytes can be swapped.
% Note: Can't swap the bytes of signed short integers in MatLab.
waveform = uint16(mod(65536 + waveform,65536)); %

% If on a PC swap the bytes to Big Endian
if strcmp( computer, 'PCWIN' )
    waveform = bitor(bitshift(waveform,-8),bitshift(waveform,8));
```

```
end

% Save the data to a file
% Note: The waveform is saved as unsigned short integers. However,
%       the acual bit pattern is that of signed short integers and
%       that is how the Agilent MXG/ESG/PSG interprets them.
filename = 'C:\Temp\PSGTestFile';
[FID, message] = fopen(filename,'w');% Open a file to write data
if FID == -1 error('Cannot Open File'); end
fwrite(FID,waveform,'unsigned short');% write to the file
fclose(FID);                          % close the file

% 3.) Load the internal Arb format file *********************************
% This process is just the reverse of saving the waveform
% Read in waveform as unsigned short integers.
% Swap the bytes as necessary
% Convert to signed integers then normalize between +-1
% De-interleave the I/Q Data

% Open the file and load the internal format data
[FID, message] = fopen(filename,'r');% Open file to read data
if FID == -1 error('Cannot Open File'); end
[internalWave,n] = fread(FID, 'uint16');% read the IQ file
fclose(FID);% close the file

internalWave = internalWave';   % Conver from column array to row array

% If on a PC swap the bytes back to Little Endian
if strcmp( computer, 'PCWIN' )  % Put the bytes into the correct order
    internalWave= bitor(bitshift(internalWave,-8),bitshift(bitand(internalWave,255),8));
end

% convert unsigned to signed representation
internalWave = double(internalWave);
tmp = (internalWave > 32767.0) * 65536;
iqWave = (internalWave - tmp) ./ 32767;  % and normalize the data

% De-Interleave the IQ data
IwaveIn = iqWave(1:2:n);
QwaveIn = iqWave(2:2:n);
```

Creating and Downloading a Pulse

NOTE This section applies only to the Agilent MXG and the PSG.

For the Agilent MXG, the maximum frequency is 6 GHz, and the *pulsepat.m* program's `SOURce:FREQuency 20000000000` value must be changed as required in the following programs. For more frequency information, refer to the signal generator's *Data Sheet*.

On the documentation CD, this programming example's name is "*pulsepat.m*."

This MATLAB programming example performs the following functions:

- I and Q data creation for 10 pulses
- marker file creation
- data scaling
- downloading using Agilent Waveform Download Assistant functions (see "Using the Download Utilities" on page 242 for more information)

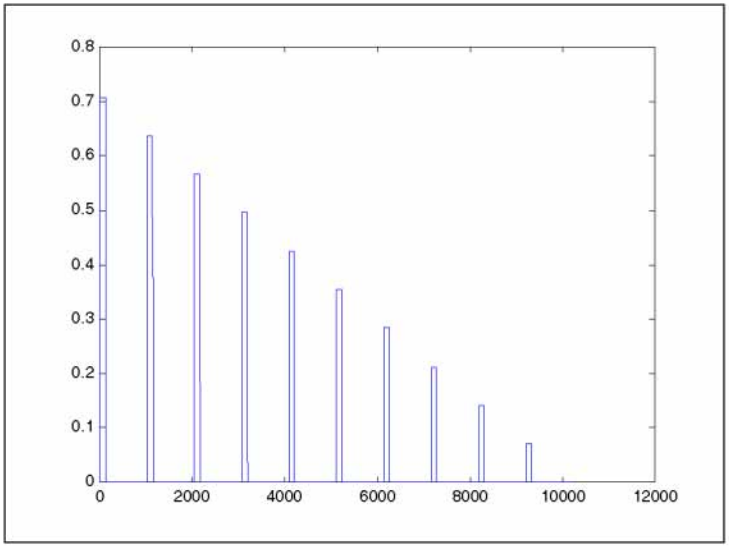
```
% Script file: pulsepat.m
%
% Purpose:
%To calculate and download an arbitrary waveform file that simulates a
%simple antenna scan pulse pattern to the Agilent MXG/PSG vector signal generator.
%
% Define Variables:
% n -- counting variable (no units)
% t -- time (seconds)
% rise -- raised cosine pulse rise–time definition (samples)
% on -- pulse on–time definition (samples)
% fall -- raised cosine pulse fall–time definition (samples)
% i -- in–phase modulation signal
% q -- quadrature modulation signal

n=4;                        % defines the number of points in the rise–time and fall–time
t=–1:2/n:1–2/n;             % number of points translated to time
rise=(1+sin(t*pi/2))/2;     % defines the pulse rise–time shape
on=ones(1,120);             % defines the pulse on–time characteristics
fall=(1+sin(–t*pi/2))/2;    % defines the pulse fall–time shape
off=zeros(1,896);           % defines the pulse off–time characteristics
```

```
% arrange the i–samples and scale the amplitude to simulate an antenna scan
% pattern comprised of 10 pulses
i = .707*[rise on fall off...
[.9*[rise on fall off]]...
[.8*[rise on fall off]]...
[.7*[rise on fall off]]...
[.6*[rise on fall off]]...
[.5*[rise on fall off]]...
[.4*[rise on fall off]]...
[.3*[rise on fall off]]...
[.2*[rise on fall off]]...
[.1*[rise on fall off]]];

% set the q–samples to all zeroes
q = zeros(1,10240);

% define a composite iq matrix for download to the Agilent MXG/PSG using the
% Waveform Download Assistant
IQData = [i + (j * q)];

% define a marker matrix and activate a marker to indicate the beginning of the waveform
Markers = zeros(2,length(IQData));        % fill marker array with zero, i.e no markers set
Markers(1,1) = 1;          % set marker to first point of playback

% make a new connection to theAgilent MXG/PSG over the GPIB interface
io = agt_newconnection('gpib',0,19);

% verify that communication with the Agilent MXG/PSG has been established
[status, status_description, query_result] = agt_query(io,'*idn?');
if (status < 0) return; end

% set the carrier frequency and power level on the Agilent MXG/PSG using the Agilent
%Waveform Download Assistant
```

```
[status, status_description] = agt_sendcommand(io, 'SOURce:FREQuency 20000000000');
[status, status_description] = agt_sendcommand(io, 'POWer 0');

% define the ARB sample clock for playback
sampclk = 40000000;

% download the iq waveform to the PSG baseband generator for playback
[status, status_description] = agt_waveformload(io, IQData, 'pulsepat', sampclk, 'play', 'no_normscale',
Markers);

% turn on RF output power
[status, status_description ] = agt_sendcommand( io, 'OUTPut:STATe ON' )
```

You can test your program by performing a simulated plot of the in–phase modulation signal in Matlab (see Figure 5-2 on page 273). To do this, enter `plot (i)` at the Matlab command prompt.

Figure 5-2 Simulated Plot of In–Phase Signal



The following additional Matlab M–file pulse programming examples are also available on the *Documentation CD–ROM* for your Agilent MXG and PSG signal generator:

NOTE For the Agilent MXG, the `SOURce:FREQuency 20000000000` value must be changed as required in the following programs. For more information, refer to the *Data Sheet*.

barker.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple 7–bit barker RADAR signal to the PSG vector signal generator.
chirp.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple compressed pulse RADAR signal using linear FM chirp to the PSG vector signal generator.
FM.m	This programming example calculates and downloads an arbitrary waveform file that simulates a single tone FM signal with a rate of 6 KHz, deviation of =/– 14.3 KHz, Bessel null of dev/rate=2.404 to the Agilent MXG/PSG vector signal generator.
nchirp.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple compressed pulse RADAR signal using non–linear FM chirp to the PSG vector signal generator.
pulse.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple pulse signal to the PSG vector signal generator.
pulsedroop.m	This programming example calculates and downloads an arbitrary waveform file that simulates a simple pulse signal with pulse droop to the PSG vector signal generator.

Visual Basic Programming Examples

Creating I/Q Data—Little Endian Order

On the documentation CD, this programming example's name is "*Create_IQData_vb.txt*."

This Visual Basic programming example, using Microsoft Visual Basic 6.0, uses little endian order data, and performs the following functions:

- error checking
- I an Q integer array creation
- I an Q data interleaving
- byte swapping to convert to big endian order
- binary data file storing to a PC or workstation

Once the file is created, you can download the file to the signal generator using FTP (see "FTP Procedures" on page 224).

```
'*************************************************************************************
' Program Name: Create_IQData
' Program Description: This program creates a sine and cosine wave using 200 I/Q data
' samples. Each I and Q value is represented by a 2 byte integer. The sample points are
' calculated, scaled using the AMPLITUDE constant of 32767, and then stored in an array
' named iq_data. The AMPLITUDE scaling allows for full range I/Q modulator DAC values.
' Data must be in 2's complemant, MSB/LSB big-endian format. If your PC uses LSB/MSB
' format, then the integer bytes must be swapped. This program converts the integer
' array values to hex data types and then swaps the byte positions before saving the
' data to the IQ_DataVB file.
'*************************************************************************************
Private Sub Create_IQData()
Dim index As Integer
Dim AMPLITUDE As Integer
Dim pi As Double
Dim loByte As Byte
Dim hiByte As Byte
Dim loHex As String
Dim hiHex As String
Dim strSrc As String
Dim numPoints As Integer
Dim FileHandle As Integer
Dim data As Byte
Dim iq_data() As Byte
Dim strFilename As String

strFilename = "C:\IQ_DataVB"

Const SAMPLES = 200     ' Number of sample PAIRS of I and Q integers for the waveform
```

```
AMPLITUDE = 32767       ' Scale the amplitude for full range of the signal generators
                        ' I/Q modulator DAC
pi = 3.141592

Dim intIQ_Data(0 To 2 * SAMPLES - 1)  'Array for I and Q integers: 400
ReDim iq_data(0 To (4 * SAMPLES - 1)) 'Need MSB and LSB bytes for each integer value: 800

'Create an integer array of I/Q pairs

    For index = 0 To (SAMPLES - 1)
        intIQ_Data(2 * index) = CInt(AMPLITUDE * Sin(2 * pi * index / SAMPLES))
        intIQ_Data(2 * index + 1) = CInt(AMPLITUDE * Cos(2 * pi * index / SAMPLES))
    Next index

 'Convert each integer value to a hex string and then write into the iq_data byte array
 'MSB, LSB ordered
 For index = 0 To (2 * SAMPLES - 1)
    strSrc = Hex(intIQ_Data(index)) 'convert the integer to a hex value

    If Len(strSrc) <> 4 Then
        strSrc = String(4 - Len(strSrc), "0") & strSrc  'Convert to hex format i.e "800F
    End If                                              'Pad with 0's if needed to get 4
                                                        'characters i.e '0' to "0000"

       hiHex = Mid$(strSrc, 1, 2)   'Get the first two hex values (MSB)
       loHex = Mid$(strSrc, 3, 2)   'Get the next two hex values (LSB)
       loByte = CByte("&H" & loHex) 'Convert to byte data type LSB
       hiByte = CByte("&H" & hiHex) 'Convert to byte data type MSB

       iq_data(2 * index) = hiByte      'MSB into first byte
       iq_data(2 * index + 1) = loByte  'LSB into second byte

  Next index

  'Now write the data to the file

FileHandle = FreeFile()     'Get a file number

numPoints = UBound(iq_data) 'Get the number of bytes in the file

Open strFilename For Binary Access Write As #FileHandle Len = numPoints + 1
```

```
On Error GoTo file_error

    For index = 0 To (numPoints)
        data = iq_data(index)
        Put #FileHandle, index + 1, data 'Write the I/Q data to the file
    Next index

Close #FileHandle

Call MsgBox("Data written to file " & strFilename, vbOKOnly, "Download")

Exit Sub

file_error:
    MsgBox Err.Description
    Close #FileHandle

End Sub
```

Downloading I/Q Data

On the signal generator's documentation CD, this programming example's name is "*Download_File_vb.txt*."

This Visual Basic programming example, using Microsoft Visual Basic 6.0, downloads the file created in "Creating I/Q Data–Little Endian Order" on page 275 into non–volatile memory using a LAN connection. To use GPIB, replace the instOpenString object declaration with "GPIB::19::INSTR". To download the data into volatile memory, change the instDestfile declaration to "USER/BBG1/WAVEFORM/".

NOTE The example program listed here uses the VISA COM IO API, which includes the WriteIEEEBlock method. This method eliminates the need to format the download command with arbitrary block information such as defining number of bytes and byte numbers. Refer to "SCPI Command Line Structure" on page 220 for more information.

This program also includes some error checking to alert you when problems arise while trying to download files. This includes checking to see if the file exists.

```
'**************************************************************************
' Program Name: Download_File
' Program Description: This program uses Microsoft Visual Basic 6.0 and the Agilent
' VISA COM I/O Library to download a waveform file to the signal generator.
'
' The program downloads a file (the previously created 'IQ_DataVB' file) to the signal
' generator. Refer to the Programming Guide for information on binary
```

```
' data requirements for file downloads. The waveform data 'IQ_DataVB' is
' downloaded to the signal generator's non-volatile memory(NVWFM)
' " /USER/WAVEFORM/IQ_DataVB". For volatile memory(WFM1) download to the
' " /USER/BBG1/WAVEFORM/IQ_DataVB" directory.
'
' You must reference the Agilent VISA COM Resource Manager and VISA COM 1.0 Type
' Library in your Visual Basic project in the Project/References menu.
' The VISA COM 1.0 Type Library, corresponds to VISACOM.tlb and the Agilent
' VISA COM Resource Manager, corresponds to AgtRM.DLL.
' The VISA COM 488.2 Formatted I/O 1.0, corresponds to the BasicFormattedIO.dll
' Use a statement such as "Dim Instr As VisaComLib.FormattedIO488" to
' create the formatted I/O reference and use
' "Set Instr = New VisaComLib.FormattedIO488" to create the actual object.
'*******************************************************************************
' IMPORTANT: Use the TCPIP address of your signal generator in the rm.Open
' declaraion. If you are using the GPIB interface in your project use "GPIB::19::INSTR"
' in the rm.Open declaration.
'*******************************************************************************

Private Sub Download_File()
' The following four lines declare IO objects and instantiate them.
Dim rm As VisaComLib.ResourceManager
Set rm = New AgilentRMLib.SRMCls
Dim SigGen As VisaComLib.FormattedIO488
Set SigGen = New VisaComLib.FormattedIO488

' NOTE: Use the IP address of your signal generator in the rm.Open declaration
Set SigGen.IO = rm.Open("TCPIP0::000.000.000.000")

Dim data As Byte
Dim iq_data() As Byte
Dim FileHandle As Integer
Dim numPoints As Integer
Dim index As Integer
Dim Header As String
Dim response As String
Dim hiByte As String
Dim loByte As String
Dim strFilename As String

strFilename = "C:\IQ_DataVB" ‘File Name and location on PC
                            'Data will be saved to the signal generator’s NVWFM
                        ‘/USER/WAVEFORM/IQ_DataVB directory.
```

```
FileHandle = FreeFile()

On Error GoTo errorhandler

With SigGen                 'Set up the signal generator to accept a download
    .IO.Timeout = 5000      'Timeout 50 seconds
    .WriteString "*RST"     'Reset the signal generator.
End With

numPoints = (FileLen(strFilename))  'Get number of bytes in the file: 800 bytes

ReDim iq_data(0 To numPoints - 1)      'Dimension the iq_data array to the
                                       'size of the IQ_DataVB file: 800 bytes

Open strFilename For Binary Access Read As #FileHandle  'Open the file for binary read
On Error GoTo file_error

For index = 0 To (numPoints - 1)    'Write the IQ_DataVB data to the iq_data array
        Get #FileHandle, index + 1, data    '(index+1) is the record number
        iq_data(index) = data
Next index

    Close #FileHandle                   'Close the file

'Write the command to the Header string. NOTE: syntax
 Header = "MEM:DATA ""/USER/WAVEFORM/IQ_DataVB""","

   'Now write the data to the signal generator's non-volatile memory (NVWFM)

    SigGen.WriteIEEEBlock Header, iq_data

    SigGen.WriteString "*OPC?"          'Wait for the operation to complete
    response = SigGen.ReadString        'Signal generator reponse to the OPC? query
    Call MsgBox("Data downloaded to the signal generator", vbOKOnly, "Download")
    Exit Sub
errorhandler:
    MsgBox Err.Description, vbExclamation, "Error Occurred", Err.HelpFile, Err.HelpContext
Exit Sub
file_error:
    Call MsgBox(Err.Description, vbOKOnly)  'Display any error message
    Close #FileHandle
End Sub
```

HP Basic Programming Examples

This section contains the following programming examples:

- "Creating and Downloading Waveform Data Using HP BASIC for Windows®" on page 280
- "Creating and Downloading Waveform Data Using HP BASIC for UNIX" on page 282
- "Creating and Downloading E443xB Waveform Data Using HP BASIC for Windows" on page 284
- "Creating and Downloading E443xB Waveform Data Using HP Basic for UNIX" on page 285

Creating and Downloading Waveform Data Using HP BASIC for Windows®

On the documentation CD, this programming example's name is "*hpbasicWin.txt*."

The following program will download a waveform using HP Basic for Windows into volatile ARB memory. The waveform generated by this program is the same as the default `SINE_TEST_WFM` waveform file available in the signal generator's waveform memory. This code is similar to the code shown for BASIC for UNIX but there is a formatting difference in line 130 and line 140.

To download into non–volatile memory, replace line 190 with:

190 OUTPUT @PSG USING "#,K";":MMEM:DATA ""NVWFM:testfile"", #"

As discussed at the beginning of this section, I and Q waveform data is interleaved into one file in 2's compliment form and a marker file is associated with this I/Q waveform file.

In the `Output` commands, `USING "#,K"` formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The `"K"` instructs HP Basic to output the following numbers or strings in the default format.

```
10 !  RE-SAVE "BASIC_Win_file"
20    Num_points=200
30    ALLOCATE INTEGER Int_array(1:Num_points*2)
40    DEG
50    FOR I=1 TO Num_points*2 STEP 2
60      Int_array(I)=INT(32767*(SIN(I*360/Num_points)))
70    NEXT I
80    FOR I=2 TO Num_points*2 STEP 2
90      Int_array(I)=INT(32767*(COS(I*360/Num_points)))
100   NEXT I
110   PRINT "Data Generated"
120   Nbytes=4*Num_points
130   ASSIGN @PSG TO 719
140   ASSIGN @PSGb TO 719;FORMAT MSB FIRST
150   Nbytes$=VAL$(Nbytes)
160   Ndigits=LEN(Nbytes$)
```

Windows and MS Windows are U.S registered trademarks of Microsoft Corporation.

```
170   Ndigits$=VAL$(Ndigits)
180   WAIT 1
190   OUTPUT @PSG USING "#,K";":MMEM:DATA ""WFM1:data_file"",#"
200   OUTPUT @PSG USING "#,K";Ndigits$
210   OUTPUT @PSG USING "#,K";Nbytes$
220   WAIT 1
230   OUTPUT @PSGb;Int_array(*)
240   OUTPUT @PSG;END
250   ASSIGN @PSG TO *
260   ASSIGN @PSGb TO *
270   PRINT
280   PRINT "*END*"
290   END
```

Program Comments

10:	Program file name
20:	Sets the number of points in the waveform.
30:	Allocates integer data array for I and Q waveform points.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up first loop for I waveform points.
60:	Calculate and interleave I waveform points.
70:	End of loop
80	Sets up second loop for Q waveform points.
90:	Calculate and interleave Q waveform points.
100:	End of loop.
120:	Calculates number of bytes in I/Q waveform.
130:	Opens an IO path to the signal generator using GPIB. 7 is the address of the GPIB card in the computer, and 19 is the address of the signal generator. This IO path is used to send ASCII data to the signal generator.
140:	Opens an IO path for sending binary data to the signal generator.
150:	Creates an ASCII string representation of the number of bytes in the waveform.
160 to 170:	Finds the number of digits in Nbytes.
190:	Sends the first part of the SCPI command, MEM:DATA along with the name of the file, `data_file`, that will receive the waveform data. The name, `data_file`, will appear in the signal generator's memory catalog.
200 to 210:	Sends the rest of the ASCII header.

Program Comments (Continued)

230:	Sends the binary data. Note that `PSGb` is the binary IO path.
240:	Sends an End–of–Line to terminate the transmission.
250 to 260:	Closes the connections to the signal generator.
290:	End the program.

Creating and Downloading Waveform Data Using HP BASIC for UNIX

On the documentation CD, this programming example's name is "*hpbasicUx.txt*."

The following program shows you how to download waveforms using HP Basic for UNIX. The code is similar to that shown for HP BASIC for Windows, but there is a formatting difference in line 130 and line 140.

To download into non–volatile memory, replace line 190 with:

190 OUTPUT @PSG USING "#,K";":MMEM:DATA ""NVWFM:testfile"", #"

As discussed at the beginning of this section, I and Q waveform data is interleaved into one file in 2's compliment form and a marker file is associated with this I/Q waveform file.

In the `Output` commands, `USING "#,K"` formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The "K" instructs HP BASIC to output the following numbers or strings in the default format.

```
10 !  RE-SAVE "UNIX_file"
20    Num_points=200
30    ALLOCATE INTEGER Int_array(1:Num_points*2)
40    DEG
50    FOR I=1 TO Num_points*2 STEP 2
60      Int_array(I)=INT(32767*(SIN(I*360/Num_points)))
70    NEXT I
80    FOR I=2 TO Num_points*2 STEP 2
90      Int_array(I)=INT(32767*(COS(I*360/Num_points)))
100   NEXT I
110   PRINT "Data generated "
120   Nbytes=4*Num_points
130   ASSIGN @PSG TO 719;FORMAT ON
140   ASSIGN @PSGb TO 719;FORMAT OFF
150   Nbytes$=VAL$(Nbytes)
160   Ndigits=LEN(Nbytes$)
170   Ndigits$=VAL$(Ndigits)
180   WAIT 1
190   OUTPUT @PSG USING "#,K";":MMEM:DATA ""WFM1:data_file"",#"
200   OUTPUT @PSG USING "#,K";Ndigits$
```

```
210   OUTPUT @PSG USING "#,K";Nbytes$
220   WAIT 1
230   OUTPUT @PSGb;Int_array(*)
240   WAIT 2
241   OUTPUT @PSG;END
250   ASSIGN @PSG TO *
260   ASSIGN @PSGb TO *
270   PRINT
280   PRINT "*END*"
290   END
```

Program Comments

10:	Program file name
20:	Sets the number of points in the waveform.
30:	Allocates integer data array for I and Q waveform points.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up first loop for I waveform points.
60:	Calculate and interleave I waveform points.
70:	End of loop
80	Sets up second loop for Q waveform points.
90:	Calculate and interleave Q waveform points.
100:	End of loop.
120:	Calculates number of bytes in I/Q waveform.
130:	Opens an IO path to the signal generator using GPIB. 7 is the address of the GPIB card in the computer, and 19 is the address of the signal generator. This IO path is used to send ASCII data to the signal generator.
140:	Opens an IO path for sending binary data to the signal generator.
150:	Creates an ASCII string representation of the number of bytes in the waveform.
160 to 170:	Finds the number of digits in Nbytes.
190:	Sends the first part of the SCPI command, MEM:DATA along with the name of the file, `data_file`, that will receive the waveform data. The name, `data_file`, will appear in the signal generator's memory catalog.
200 to 210:	Sends the rest of the ASCII header.
230:	Sends the binary data. Note that `PSGb` is the binary IO path.
240:	Sends an End–of–Line to terminate the transmission.

Program Comments (Continued)

250 to 260:	Closes the connections to the signal generator.
290:	End the program.

Creating and Downloading E443xB Waveform Data Using HP BASIC for Windows

On the documentation CD, this programming example's name is "*e443xb_hpbasicWin2.txt*."

The following program shows you how to download waveforms using HP Basic for Windows into volatile ARB memory. This program is similar to the following program example as well as the previous examples. The difference between BASIC for UNIX and BASIC for Windows is the way the formatting, for the most significant bit (MSB) on lines 110 and 120, is handled.

To download into non–volatile ARB memory, replace line 160 with:

160 OUTPUT @ESG USING "#,K";":MMEM:DATA ""NVARBI:testfile"", #"

and replace line 210 with:

210 OUTPUT @ESG USING "#,K";":MMEM:DATA ""NVARBQ:testfile"", #"

First, the I waveform data is put into an array of integers called `Iwfm_data` and the Q waveform data is put into an array of integers called Qwfm_data. The variable `Nbytes` is set to equal the number of bytes in the I waveform data. This should be twice the number of integers in `Iwfm_data`, since an integer is 2 bytes. Input integers must be between 0 and 16383.

In the `Output` commands, `USING "#,K"` formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The `"K"` instructs HP Basic to output the following numbers or strings in the default format.

```
10    ! RE-SAVE "ARB_IQ_Win_file"
20    Num_points=200
30    ALLOCATE INTEGER Iwfm_data(1:Num_points),Qwfm_data(1:Num_points)
40    DEG
50    FOR I=1 TO Num_points
60      Iwfm_data(I)=INT(8191*(SIN(I*360/Num_points))+8192)
70      Qwfm_data(I)=INT(8191*(COS(I*360/Num_points))+8192)
80    NEXT I
90    PRINT "Data Generated"
100   Nbytes=2*Num_points
110   ASSIGN @Esg TO 719
120   !ASSIGN @Esgb TO 719;FORMAT MSB FIRST
130   Nbytes$=VAL$(Nbytes)
140   Ndigits=LEN(Nbytes$)
150   Ndigits$=VAL$(Ndigits)
160   OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBI:file_name_1"",#"
170   OUTPUT @Esg USING "#,K";Ndigits$
180   OUTPUT @Esg USING "#,K";Nbytes$
```

```
190   OUTPUT @Esgb;Iwfm_data(*)
200   OUTPUT @Esg;END
210   OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBQ:file_name_1"",#"
220   OUTPUT @Esg USING "#,K";Ndigits$
230   OUTPUT @Esg USING "#,K";Nbytes$
240   OUTPUT @Esgb;Qwfm_data(*)
250   OUTPUT @Esg;END
260   ASSIGN @Esg TO *
270   ASSIGN @Esgb TO *
280   PRINT
290   PRINT "*END*"
300   END
```

Program Comments

10:	Program file name.
20	Sets the number of points in the waveform.
30:	Defines arrays for I and Q waveform points. Sets them to be integer arrays.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up loop to calculate waveform points.
60:	Calculates I waveform points.
70:	Calculates Q waveform points.
80:	End of loop.
160 and 210:	The I and Q waveform files have the same name
90 to 300:	See the table on page 281 for program comments.

Creating and Downloading E443xB Waveform Data Using HP Basic for UNIX

On the documentation CD, this programming example's name is "*e443xb_hpbasicUx2.txt*."

The following program shows you how to download waveforms using HP BASIC for UNIX. It is similar to the previous program example. The difference is the way the formatting for the most significant bit (MSB) on lines is handled.

First, the I waveform data is put into an array of integers called `Iwfm_data` and the Q waveform data is put into an array of integers called `Qwfm_data`. The variable `Nbytes` is set to equal the number of bytes in the I waveform data. This should be twice the number of integers in `Iwfm_data`, since an integer is represented 2 bytes. Input integers must be between 0 and 16383.

In the `Output` commands, `USING "#,K"` formats the data. The pound symbol (#) suppresses the automatic EOL (End of Line) output. This allows multiple output commands to be concatenated as if they were a single output. The "K" instructs HP BASIC to output the following numbers or strings in the default format.

```
10    ! RE-SAVE "ARB_IQ_file"
20    Num_points=200
30    ALLOCATE INTEGER Iwfm_data(1:Num_points),Qwfm_data(1:Num_points)
40    DEG
50    FOR I=1 TO Num_points
60      Iwfm_data(I)=INT(8191*(SIN(I*360/Num_points))+8192)
70      Qwfm_data(I)=INT(8191*(COS(I*360/Num_points))+8192)
80    NEXT I
90    PRINT "Data Generated"
100   Nbytes=2*Num_points
110   ASSIGN @Esg TO 719;FORMAT ON
120   ASSIGN @Esgb TO 719;FORMAT OFF
130   Nbytes$=VAL$(Nbytes)
140   Ndigits=LEN(Nbytes$)
150   Ndigits$=VAL$(Ndigits)
160   OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBI:file_name_1"",#"
170   OUTPUT @Esg USING "#,K";Ndigits$
180   OUTPUT @Esg USING "#,K";Nbytes$
190   OUTPUT @Esgb;Iwfm_data(*)
200   OUTPUT @Esg;END
210   OUTPUT @Esg USING "#,K";":MMEM:DATA ""ARBQ:file_name_1"",#"
220   OUTPUT @Esg USING "#,K";Ndigits$
230   OUTPUT @Esg USING "#,K";Nbytes$
240   OUTPUT @Esgb;Qwfm_data(*)
250   OUTPUT @Esg;END
260   ASSIGN @Esg TO *
270   ASSIGN @Esgb TO *
280   PRINT
290   PRINT "*END*"
300   END
```

Program Comments

10:	Program file name.
20	Sets the number of points in the waveform.
30:	Defines arrays for I and Q waveform points. Sets them to be integer arrays.
40:	Sets HP BASIC to use degrees for cosine and sine functions.
50:	Sets up loop to calculate waveform points.
60:	Calculates I waveform points.
70:	Calculates Q waveform points.

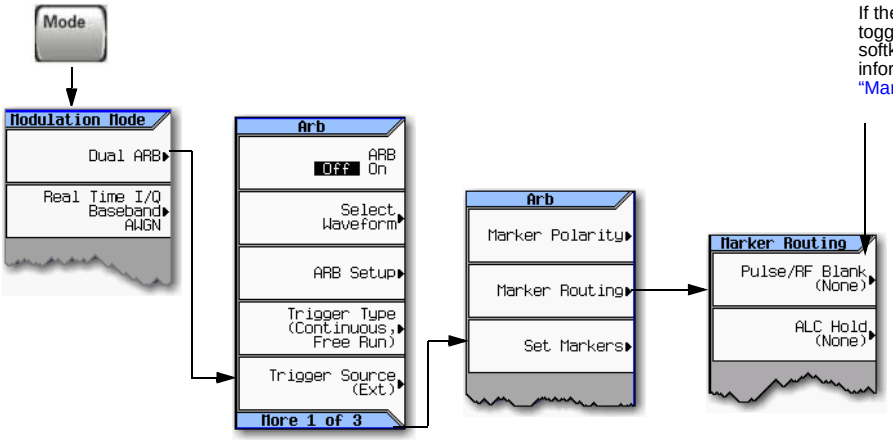
Program Comments (Continued)

80:	End of loop.
160 and 210:	The I and Q waveform files have the same name
90 to 300	See the table on page 283 for program comments.

Troubleshooting Waveform Files

Symptom	Possible Cause
ERROR 224, Text file busy	Attempting to download a waveform that has the same name as the waveform currently being played by the signal generator. To solve the problem, either change the name of the waveform being downloaded or turn off the ARB.
ERROR 628, DAC over range	The amplitude of the signal exceeds the DAC input range. The typical causes are unforeseen overshoot (DAC values within range) or the input values exceed the DAC range. To solve the problem, scale or reduce the DAC input values. For more information, see "DAC Input Values" on page 202. On the Agilent MXG, this error can occur if an encrypted file (`.SECUREWAVE`) is being downloaded to the signal generator from a PC or USB Media with a different suffix (i.e. *not* .SECUREWAVE). To solve the problem, use the **Use as** or **Copy File to Instrument** softkey menus to download the encrypted file to the instrument. For more information, see "Encrypted I/Q Files and the Securewave Directory (Agilent MXG)" on page 219.
ERROR 629, File format invalid	The signal generator requires a minimum of 60 samples to build a waveform and the same number of I and Q data points.
ERROR –321, Out of memory	There is not enough space in the ARB memory for the waveform file being downloaded. To solve the problem, either reduce the file size of the waveform file or delete unnecessary files from ARB memory. Refer to "Waveform Memory" on page 212.
No RF Output	The marker RF blanking function may be active. To check for and turn RF blanking off, refer to "Configuring the Pulse/RF Blank (Agilent MXG)" on page 288 and "Configuring the Pulse/RF Blank (ESG/PSG)" on page 288. This problem occurs when the file header contains unspecified settings and a previously played waveform used the marker RF blanking function. For more information on the marker functions, see the *User's Guide*.
Undesired output signal	Check for the following: • The data was downloaded in little endian order. See "Little Endian and Big Endian (Byte Order)" on page 200 for more information. • The waveform contains an odd number of samples. An odd number of samples can cause waveform discontinuity. See "Waveform Phase Continuity" on page 209 for more information.

Configuring the Pulse/RF Blank (Agilent MXG)



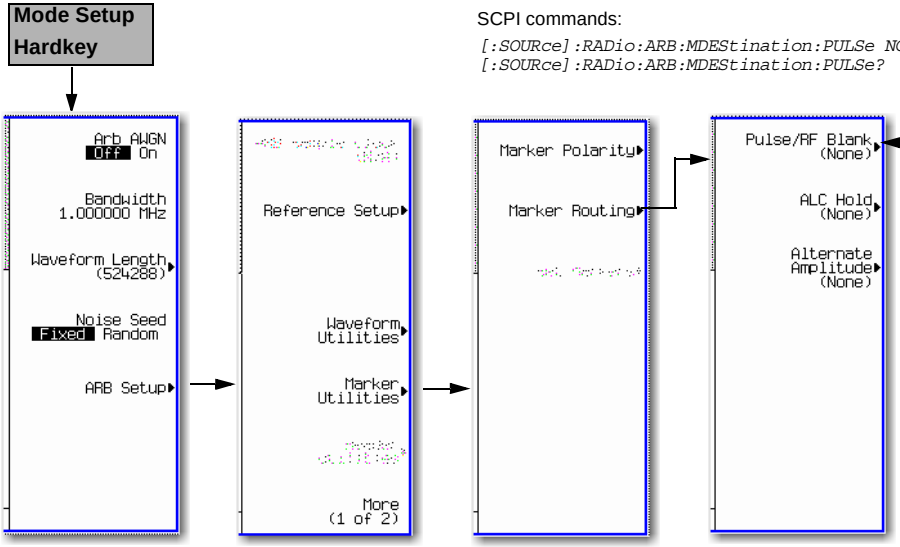
If the default marker is used, toggle the Pulse/RF Blank (None) softkey to None. For more information on markers, refer to "Marker File" on page 207.

SCPI commands:

[:SOURce]:RADio[1]:ARB:MDEStination:PULSe NONE|M1|M2|M3|M4
[:SOURce]:RADio[1]:ARB:MDEStination:PULSe?

For details on each key, use the key help. Refer to "Getting Help (Agilent MXG)" on page 18 and the *User's Guide*. For additional SCPI command information, refer to the SCPI Command Reference.

Configuring the Pulse/RF Blank (ESG/PSG)



SCPI commands:

[:SOURce]:RADio:ARB:MDEStination:PULSe NONE|M1|M2|M3|M4
[:SOURce]:RADio:ARB:MDEStination:PULSe?

If the default marker is used, toggle the Pulse/RF Blank (None) softkey to None. For more information on markers, refer to "Marker File" on page 207.

For details on each key, use the *Key and Data Field Reference*. For additional SCPI command information, refer to the SCPI Command Reference.

6 Creating and Downloading User–Data Files

NOTE Some features apply to only the E4438C with Option 001, 002, 601, or 602 and E8267D with Option 601 or 602. These exceptions are indicated in the sections.

On the Agilent MXG, the internal baseband generator speed upgrade Options 670, 671, and 672 are option upgrades that *require* Option 651 and 652 to have been loaded at the factory (refer to the *Data Sheet* for more information). Any references to 651, 652, or 654 are inclusive of 671, 672, and 674.

The following sections and procedures contain remote SCPI commands. For front panel key commands, refer to the *User's Guide*, *Key and Data Field Reference* (ESG, PSG, and E8663B), or to the Key Help in the signal generator.

For the N5161A/62A the softkey menus and features mentioned in this chapter are only available through the Web–Enabled MXG or via SCPI commands. Refer to "Using the Web Browser" on page 11 and to the *SCPI Command Reference*.

This chapter explains the requirements and processes for creating and downloading user–data, and contains the following sections:

- User File Data (Bit/Binary) Downloads (E4438C and E8267D) on page 297
- Pattern RAM (PRAM) Data Downloads (E4438C and E8267D) on page 320
- FIR Filter Coefficient Downloads (N5162A, N5182A, E4438C and E8267D) on page 333
- Save and Recall Instrument State Files on page 336
- User Flatness Correction Downloads Using C++ and VISA on page 347
- Data Transfer Troubleshooting (N5162A, N5182A, E4438C and E8267D Only) on page 351

Overview

User data is a generic term for various data types created by the user and stored in the signal generator. This includes the following data (file) types:

Bit	This file type lets the user download payload data for use in streaming or framed signals. It lets the user determine how many bits in the file the signal generator uses.
Binary	This file type provides payload data for use in streaming or framed signals. It differs from the bit file type in that you cannot specify a set number of bits. Instead the signal generator uses all bits in the file for streaming data and all bits that fill a frame for framed data. If there are not enough bits to fill a frame, the signal generator truncates the data and repeats the file from the beginning.
PRAM	With this file type, the user provides the payload data along with the bits to control signal attributes such as bursting. This file type is available for only the real–time Custom and TDMA modulation formats.
FIR Filter	This file type stores user created custom filters.
State	This file type lets the user store signal generator settings, which can be recalled. This provides a quick method for reconfiguring the signal generator when switching between different signal setups.
User Flatness Correction	This file type lets the user store amplitude corrections for frequency.

Prior to creating and downloading files, you need to take into consideration the file size and the amount of remaining signal generator memory. For more information, see “Signal Generator Memory” on page 291

Signal Generator Memory

The signal generator provides two types of memory, volatile and non–volatile.

NOTE User BIT, and User PRAM references are only applicable to the E4438C with Options 001, 002, 601, or 602, and E8267D with Options 601 or 602.

User FIR references are only applicable to the N5162A and N5182A with Options 651, 652, or 654, E4438C with Options 001, 002, 601, or 602, and E8267D with Options 601 or 602.

Volatile — Random access memory that does not survive cycling of the signal generator power. This memory is commonly referred to as waveform memory (WFM1) or pattern RAM (PRAM). Refer to Table 7 for the file types that share this memory:

Table 7 Signal Generators and Volatile Memory File Types

Volatile Memory Type	Model of Signal Generator			
	N5162A N5182A with Option 651, 652, or 654	E4438C with Option 001, 002, 601, or 602	E8267D Option 601 or 602	All Other models[a]
I/Q	x	x	x	–
Marker	x	x	x	–
File header	x	x	x	–
User PRAM	–	x	x	–
User Binary	x	x	x	–
User Bit	–	x	x	–
Waveform Sequences (multiple I/Q files played together)	n/a[b]	n/a[b]	n/a[b]	–

a. N5161A, N5181A, N5183A, E8663B, E4428C, and the E8257D.
b. Waveform sequences are always in non–volatile memory.

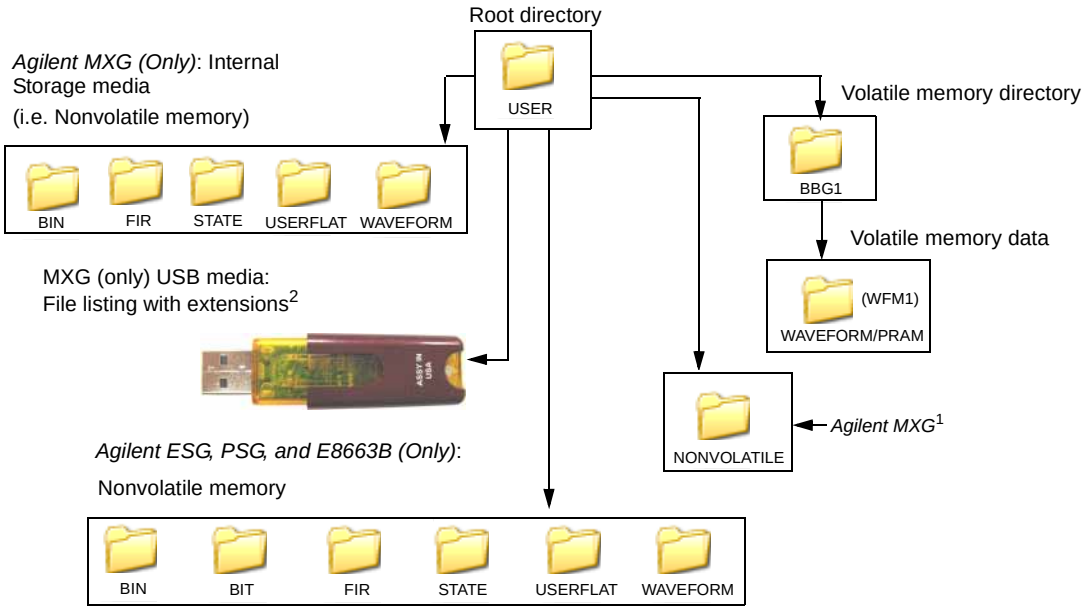
Non–volatile — Storage memory where files survive cycling of the signal generator power. Files remain until overwritten or deleted. Refer to Table 7-1 on page 292 for the file types that share this memory:

Table 7-1 Signal Generators and Non–Volatile Memory Types

Non–Volatile Memory Type	Model of Signal Generator			
	N5162A **N5182A with Option 651, 652, or 654**	**E4438C with Option 001, 002, 601, or 602**	**E8267D Option 601 or 602**	**All Other models**[a]
I/Q	x	x	x	–
Marker	x	x	x	–
File header	x	x	x	–
Sweep List	x	x	x	–
User PRAM	–	x	x	–
User Binary	x	x	x	–
User Bit	–	x	x	–
User FIR	x	x	x	–
Instrument State	x	x	x	x
Waveform Sequences (multiple I/Q files played together)	x	x	x	–

a. N5161A, N5181A, N5183A, E8663B, E4428C, and the E8257D.

The following figure shows the signal generator's directory structure for the user–data files.



[1]This NONVOLATILE directory shows the files with the same extensions as the USB media and is useful with ftp.
[2]The Agilent MXG uses an optional "USB media" to store non–volatile waveform data.

Memory Allocation

Volatile Memory

The signal generator allocates volatile memory in blocks of 1024 bytes. For example, a user–data file with 60 bytes uses 1024 bytes of memory. For a file that is too large to fit into 1024 bytes, the signal generator allocates additional memory in multiples of 1024 bytes. For example, the signal generator allocates 3072 bytes of memory for a file with 2500 bytes.

3 x 1024 bytes = 3072 bytes of memory

As shown in the examples, files can cause the signal generator to allocate more memory than what is actually used, which decreases the amount of available memory.

User–data blocks consist of 1024 bytes of memory. Each user–data file has a file header that uses 512 bytes for the Agilent MXG, or 256 bytes for the ESG/PSG in the first data block for each user–data file.

Non–Volatile Memory (Agilent MXG)

On the N5182A, non–volatile files are stored on the non–volatile internal signal generator memory (i.e. internal storage) or to the USB media, if available. The Agilent MXG non–volatile internal memory allocated according to a Microsoft compatible file allocation table (FAT) file system. The Agilent MXG signal generator allocates non–volatile memory in clusters according to the drive size

(see table on page 294). For example, referring to table on page 294, if the drive size is 15 MB and if the file is less than or equal to 4k bytes, the file uses only one 4 KB cluster of memory. For files larger than 4 KB, and with a drive size of 15 MB, the signal generator allocates additional memory in multiples of 4KB clusters. For example, a file that has 21,538 bytes consumes 6 memory clusters (24,000 bytes).

On the Agilent MXG the non–volatile memory is *also* referred to as internal storage and USB media. The Internal and USB media files /USERS/NONVOLATILE Directory contains file names with full extensions (i.e. .marker, .header, etc.).

For more information on default cluster sizes for FAT file structures, refer to Table 7-2 on page 294 and to *http://support.microsoft.com/*.

Table 7-2

Drive Size (logical volume)	Cluster Size (Bytes) (Minimum Allocation Size)
0 MB – 15 MB	4K
16 MB – 127 MB	2K
128 MB – 255 MB	4K
256 MB – 511 MB	8K
512 MB – 1023 MB	16k
1024 MB – 2048 MB	32K
2048 MB – 4096 MB	64K
4096 MB – 8192 MB	128K
8192 MB – 16384 MB	256K

Non–Volatile Memory (ESG, PSG, and E8663B)

The signal generator allocates non–volatile memory in blocks of 512 bytes. For files less than or equal to 512 bytes, the file uses only one block of memory. For files larger than 512 bytes, the signal generator allocates additional memory in multiples of 512 byte blocks. For example, a file that has 21,538 bytes consumes 43 memory blocks (22,016 bytes).

Memory Size

For the E4438C, E8267D, and E8663B, the maximum volatile memory size for user data is less than the maximum size for waveform files. This is because the signal generator permanently allocates a portion of the volatile memory for waveform markers. The values in Table 7-3 is the total amount of memory after deducting the waveform marker memory allocation.

The amount of available memory, volatile and non–volatile, varies by signal generator option and the size of the other files that share the memory. The baseband generator (BBG) options contain the volatile memory. Table 7-3 shows the maximum available memory assuming that there are no other files residing in memory.

Table 7-3 Maximum Signal Generator Memory

Volatile (WFM1/PRAM) Memory			Non–Volatile (NVWFM) Memory	
Option	**Size**		**Option**	**Size**
N5162A and N5182A				
651, 652, 654[a] (BBG)	40 MB		**Standard**	4 GB[b]
019	320 MB		**USB memory stick**	*user determined*
E4438C and E8267D				
001, 601 (BBG)[c]	32 MB		**Standard**	512 MB
002 (BBG)[c]	128 MB		**005 (Hard disk)**	6 GB
602 (BBG)	256 MB		----	----

a. The internal baseband generator speed upgrade Options 670, 671, and 672 are option upgrades that *require* Option 651 and 652 to have been loaded at the factory (refer to the *Data Sheet* for more information). Any references to 651, 652, or 654 are inclusive of 671, 672, and 674.
b. For serial numbers <MY4818xxxx, US4818xxxx, and SG4818xxxx, the persistent memory value = 512 MB.
c. Options 001 and 002 apply to only the E4438C ESG.

Checking Available Memory

Whenever you download a user–data file, you must be aware of the amount of remaining signal generator memory. Table 7-4 shows to where each user–data file type is downloaded and from which memory type the signal generator accesses the file data. Information on downloading a user–data file is located within each user–data file section.

NOTE The Bit, PRAM, and State user–data (file) types only apply to the E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

The FIR filter, (file) types only apply to the N5162A and N5182A with Option 651, 652, or 654, E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

Table 7-4 User–Data File Memory Location

User–Data File Type	Download Memory	Access Memory
Bit	Non–volatile	Volatile
Binary	Non–volatile	Volatile
PRAM	Volatile	Volatile
Instrument State	Non–volatile	Non–volatile
FIR	Non–volatile	Non–volatile
Flatness	Non–volatile	Non–volatile

Bit and binary files increase in size when the signal generator loads the data from non–volatile to volatile memory. For more information, see "User File Size" on page 302.

Use the following SCPI commands to determine the amount of remaining memory:

Volatile Memory `:MMEM:CAT? "WFM1"`

The query returns the following information:

```
<memory used>,<memory remaining>,<"file_names">
```

Non–Volatile Memory `:MEM:CAT:ALL?`

The query returns the following information:

```
<memory used>,<memory remaining>,<"file_names">
```

NOTE The signal generator calculates the memory values based on the number of bytes used by the files residing in volatile or non–volatile memory, and not on the memory block allocation. To accurately determine the available memory, you must calculate the number of blocks of memory used by the files. For more information on memory block allocation, see "Memory Allocation" on page 293.

User File Data (Bit/Binary) Downloads (E4438C and E8267D)

NOTE This section applies only to the E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

If you encounter problems with this section, refer to "Data Transfer Troubleshooting (N5162A, N5182A, E4438C and E8267D Only)" on page 351.

To verify the SCPI parser's responsiveness when remotely using the :MEM:DATA SCPI command to upload files, the file's upload should be verified using the *STB? command. Refer to the *SCPI Command Reference*.

The signal generator accepts externally created and downloaded user file data for real–time modulation formats that have user file as a data selection (shown as <"file_name"> in the data selection SCPI command). When you select a user file, the signal generator incorporates the user file data (payload data) into the modulation format's data fields. You can create the data using programs such as MATLAB or Mathcad. The following table shows the available real–time modulation formats by signal generator model:

E4438C ESG	
CDMA[a]	TDMA[b]
Custom[c]	W–CDMA[d]
GPS[e]	---

E2867D PSG
Custom[c]

a. Requires Option 401.
b. Real–time TDMA modulation formats require Option 402 and include EDGE, GSM, NADC, PDC, PHS, DECT, and TETRA.
c. For ESG, requires Option 001, 002, 601, or 602, for PSG requires Option 601 or 602.
d. Requires Option 400.
e. Requires Option 409.

The signal generator uses two file types for downloaded user file data: bit and binary. With a bit file, the signal generator views the data up to the number of bits specified when the file was downloaded. For example, if you specify to use 153 bits from a 160 bit (20 bytes) file, the signal generator transmits 153 bits and ignores the remaining 7 bits. This provides a flexible means in which to control the number of transmitted data bits. It is the preferred file type and the easiest one to use.

With a binary file, the signal generator sees all bytes (bits) in a downloaded file and attempts to use them. This can present challenges especially when working with framed data. In this situation, your file needs to contain enough bits to fill a frame or timeslot, or multiple frames or timeslots, to end on the desired boundary. To accomplish this, you may have to remove or add bytes. If there are not enough bits remaining in the file to fill a frame or timeslot, the signal generator truncates the data causing a discontinuity in the data pattern.

You download a user file to either the Bit or Binary memory catalog (directory). Unlike a PRAM file

(covered later in this chapter), user file data does not contain control bits, it is just data. The signal generator adds control bits to the user file data when it generates the signal. There are two ways that the signal generator uses the data, either in a continuous data pattern (unframed) or within framed boundaries. Real–time Custom uses only unframed data, real–time TDMA modulation formats use both types, and the others use only framed data.

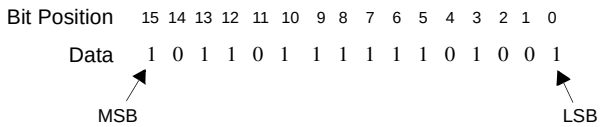
NOTE For unframed data transmission, the signal generator requires a minimum of 60 symbols. For more information, see "Determining Memory Usage for Custom and TDMA User File Data" on page 303.

You create the user file to either fill a single timeslot/frame or multiple timeslots/frames. To create multiple timeslots/frames, simply size the file with enough data to fill the number of desired timeslots/frames

User File Bit Order (LSB and MSB)

The signal generator views the data from the most significant bit (MSB) to the least significant bit (LSB). When you create your user file data, it is important that you organize the data in this manner. Within groups (strings) of bits, a bit's value (significance) is determined by its location in the string. The following shows an example of this order using two bytes.

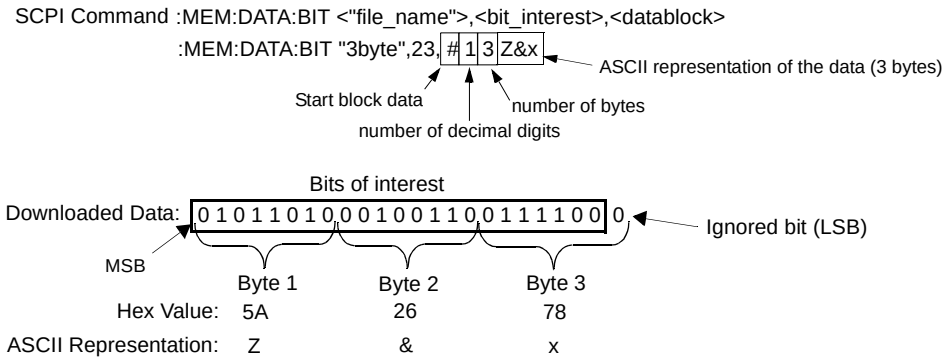
Most Significant Bit (MSB)	This bit has the highest value (greatest weight) and is located at the far left of the bit string.
Least Significant Bit (LSB)	This bit has the lowest value (bit position zero) and is located at the far right of the bit string.



Bit File Type Data

The bit file is the preferred file type and the easiest to use. When you download a bit file, you designate how many bits in the file the signal generator can modulate onto the signal. During the file download, the signal generator adds a 10–byte file header that contains the information on the number of bits the signal generator is to use.

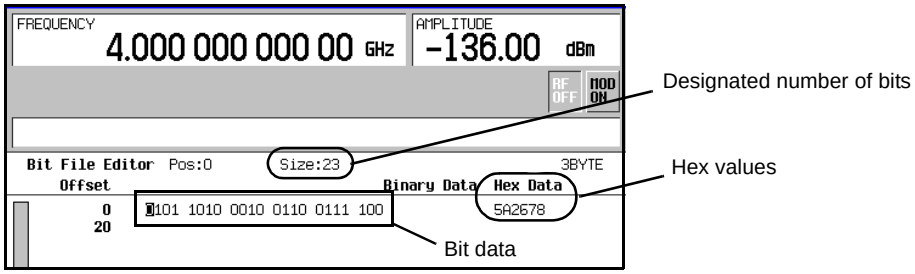
Although you download the data in bytes, when the signal generator uses the data, it recognizes only the bits of interest that you designate in the SCPI command and ignores the remaining bits. This provides greater flexibility in designing a data pattern without the concern of using an even number of bytes as is needed with the binary file data format. The following figure illustrates this concept. The example in the figure shows the bit data SCPI command formatted to download three bytes of data, but only 23 bits of the three bytes are designated as the bits of interest. (For more information on the bit data SCPI command format, see "Downloading User Files" on page 306 and "Command for Bit File Downloads" on page 309.)


SCPI Command :MEM:DATA:BIT <"file_name">,<bit_interest>,<datablock>
:MEM:DATA:BIT "3byte",23, # 1 3 Z&x
ASCII representation of the data (3 bytes)
Start block data
number of bytes
number of decimal digits
Bits of interest
Downloaded Data: 0 1 0 1 1 0 1 0 0 0 1 0 0 1 1 0 0 1 1 1 1 0 0 0
Ignored bit (LSB)
MSB
Byte 1
Byte 2
Byte 3
Hex Value: 5A 26 78
ASCII Representation: Z & x

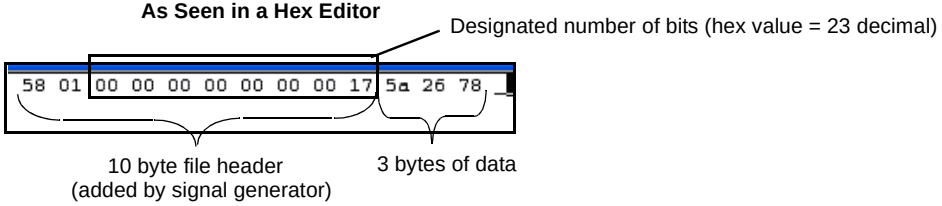
The following figure shows the same downloaded data from the above example as viewed in the signal generator's bit file editor (see the *User's Guide* for more information) and with using an external hex editor program.

SCPI command to download the data :MEM:DATA:BIT "3byte",23,#13Z&x

As Seen in the Signal Generator's Bit File Editor



As Seen in a Hex Editor

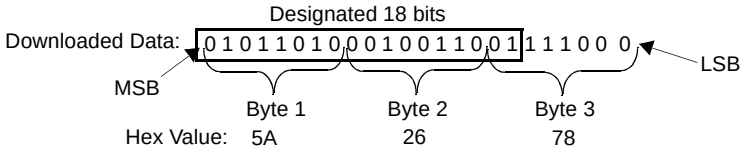

Designated number of bits (hex value = 23 decimal)
58 01 00 00 00 00 00 00 00 17 5a 26 78
10 byte file header
(added by signal generator)
3 bytes of data

In the bit editor, notice that the ignored bit of the bit–data is not displayed, however the hex value still shows all three bytes. This is because bits 1 through 7 are part of the first byte, which is shown as ASCII character x in the SCPI command line. The view from the hex editor program confirms that the downloaded three bytes of data remains unchanged. To view a downloaded bit file with an external hex editor program, FTP the file to your PC/UNIX workstation. For information on how to FTP a file, see "FTP Procedures" on page 313.

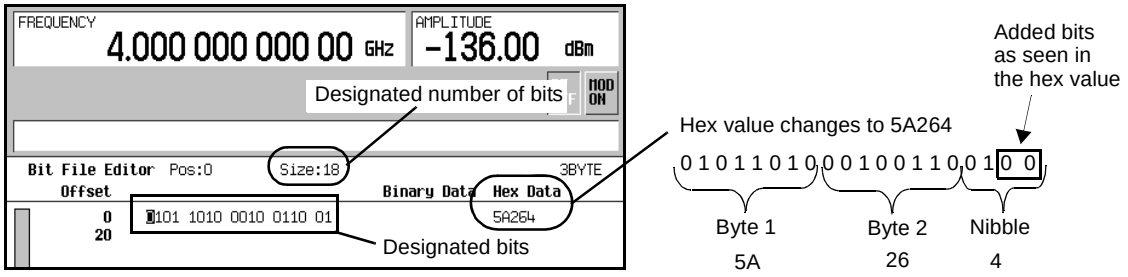
Even though the signal generator views the downloaded data on a bit basis, it groups the data into bytes, and when the designated number of bits is not a multiple of 8 bits, the last byte into one or

more 4–bit nibbles. To make the last nibble, the signal generator adds bits with a value of zero. The signal generator does not show the added bits in the bit editor and ignores the added bits when it modulates the data onto the signal, but these added bits do appear in the hex value displayed in the bit file editor. The following example, which uses the same three bytes of data, further demonstrates how the signal generator displays the data when only two bits of the last byte are part of the bits of interest.

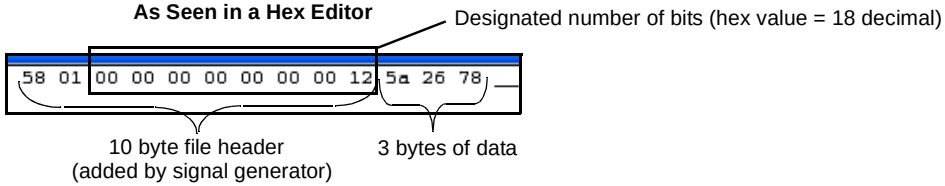
SCPI command to download the data :MEM:DATA:BIT "3byte",18,#13Z&x



As Seen in the Signal Generator's Bit File Editor



As Seen in a Hex Editor


Designated number of bits (hex value = 18 decimal)
58 01 00 00 00 00 00 00 00 12 5a 26 78
10 byte file header
(added by signal generator)
3 bytes of data

Notice that the bit file editor shows only two bytes and one nibble. In addition, the signal generator shows the nibble as hex value 4 instead of 7 (78 is byte 3—ASCII character x in the SCPI command line). This is because the signal generator sees bits 17 and 18, and assumes bits 19 and 20 are 00. As viewed by the signal generator, this makes the nibble 0100. Even though the signal generator extrapolates bits 19 and 20 to complete the nibble, it ignores these bits along with bits 21 through 24. As seen with the hex editor program, the signal generator does not actually change the three bytes of data in the downloaded file.

For information on editing a file after downloading, see "Modifying User File Data" on page 312.

Binary File Type Data

With the Binary file type, the signal generator sees all of the bytes within the downloaded file and attempts to use all of the data bits. When using this file type, the biggest challenge is creating the data, so that the signal generator uses all of the bits (bytes) contained within the file. This is referred to as using an even number of bytes. The method of creating the user file data pattern depends on whether you are using unframed or framed data. The following two sections illustrate the complexities of using the binary file format. You can eliminate these complexities by using the bit file format (see "Bit File Type Data" on page 298).

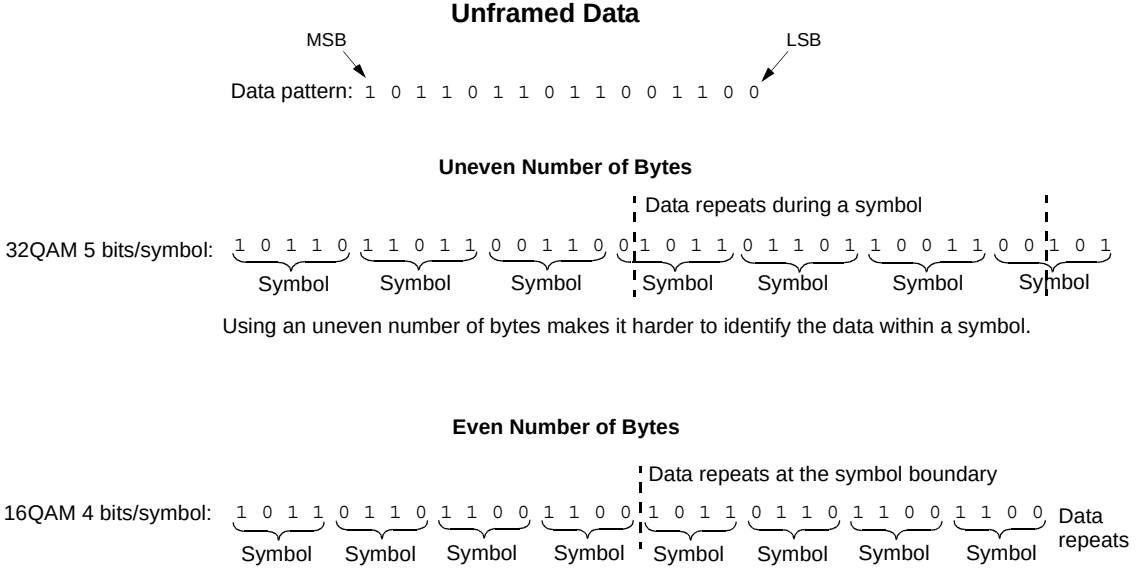
Unframed Binary Data

When creating unframed data, you must think in terms of bits per symbol; so that your data pattern begins and ends on the symbol boundary, with an even number of bytes. For example, to use 16QAM modulation, the user file needs to contain 32 bytes:

- enough data to fill 16 states 4 times
- end on a symbol boundary
- create 64 symbols (the signal generator requires a minimum of 60 symbols for unframed data)

To do the same with 32QAM, requires a user file with 40 bytes.

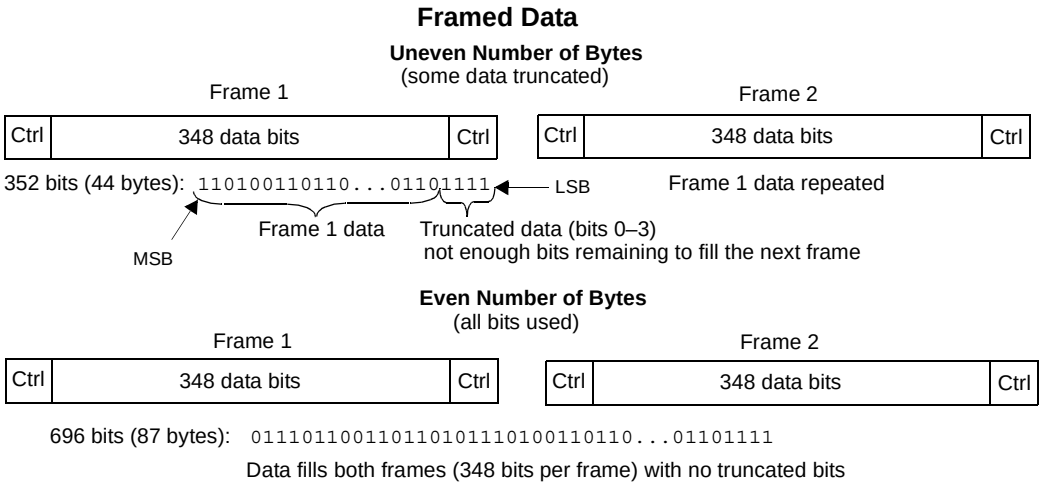
When you do *not* use an even number of bytes, the signal generator repeats the data in the same symbol where the data stream ends. This means that your data would not end on the symbol boundary, but during a symbol. This makes it harder to identify the data content of a symbol. The following figure illustrates the use of an uneven number of bytes and an even number of bytes.


Unframed Data
MSB
LSB
Data pattern: 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0
Uneven Number of Bytes
Data repeats during a symbol
32QAM 5 bits/symbol:
1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0 1 0 1
Symbol
Symbol
Symbol
Symbol
Symbol
Symbol
Symbol
Using an uneven number of bytes makes it harder to identify the data within a symbol.
Even Number of Bytes
Data repeats at the symbol boundary
16QAM 4 bits/symbol:
1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 0
Symbol
Symbol
Symbol
Symbol
Symbol
Symbol
Symbol
Symbol
Data repeats

Framed Binary Data

When using framed data, ensure that you use an even number of bytes and that the bytes contain enough bits to fill the data fields within a timeslot or frame. When there are not enough bits to fill a single timeslot or frame, the signal generator replicates the data pattern until it fills the timeslot/frame.

The signal generator creates successive timeslots/frames when the user file contains more bits than what it takes to fill a single timeslot or frame. When there are not enough bits to completely fill successive timeslots or frames, the signal generator truncates the data at the bit location where there is not enough bits remaining and repeats the data pattern. This results in a data pattern discontinuity. For example, a frame structure that uses 348 data bits requires a minimum file size of 44 bytes (352 bits), but uses only 43.5 bytes (348 bits). In this situation, the signal generator truncates the data from bit 3 to bit 0 (bits in the last byte). Remember that the signal generator views the data from MSB to LSB. For this example to have an even number of bytes and enough bits to fill the data fields, the file needs 87 bytes (696 bits). This is enough data to fill two frames while maintaining the integrity of the data pattern, as illustrated in the following figure.



For information on editing a file after downloading, see "Modifying User File Data" on page 312.

User File Size

You download user files into non–volatile memory. For CDMA, GPS, and W–CDMA, the signal generator accesses the data directly from non–volatile memory, so the file size up to the maximum file size (shown in Table 7-5) for these formats is limited only by the amount of available non–volatile memory. As seen in the table, the baseband generator option does not affect these file sizes.

For Custom and TDMA, however, when the signal generator creates the signal, it loads the data from non–volatile memory into volatile memory, which is also the same memory that the signal generator uses for Arb–based waveforms. For user data files, volatile memory is commonly referred to as pattern ram memory (PRAM). Because the Custom and TDMA user files use volatile memory, their maximum file size depends on the baseband generator (BBG) option and the amount of available

PRAM. (Volatile memory resides on the BBG.) Table 7-5 shows the maximum file sizes by modulation format and baseband generator option.

Table 7-5 Maximum User File Size

Modulation Format	Baseband Generator Option		
	001, 601	002	602
Custom[a] TDMA[a]	800 kB	3.2 MB	6.4 MB
CDMA[b] GPS[b] W–CDMA[b]	10 kB	10 kB	10 kB

a. File size with no other files residing in volatile memory.
b. File size is not affected by the BBG option.

For more information on signal generator memory, see "Signal Generator Memory" on page 291. To determine how much memory is remaining in non–volatile and volatile memory, see "Checking Available Memory" on page 295.

Determining Memory Usage for Custom and TDMA User File Data

For Custom and TDMA user files, the signal generator uses both non–volatile and volatile (PRAM/waveform) memory: you download the user file to non–volatile memory. To determine if there is enough non–volatile memory, check the available non–volatile memory and compare it to the size of the file to be downloaded.

After you select a user file and turn the format on, the signal generator loads the file into volatile memory for processing:

- It translates each data bit into a 32–bit word (4 bytes).

  The 32–bit words are not saved to the original file that resides in non–volatile memory.
- It creates an expanded data file named AUTOGEN_PRAM_1 in volatile memory while also maintaining a copy of the original file in volatile memory. It is the AUTOGEN_PRAM_1 file that contains the 32–bit words and accounts for most of the user file PRAM memory space.
- If the transmission is using unframed data and there are not enough bits in the data file to create 60 symbols, the signal generator replicates the data pattern until there is enough data for 60 symbols. For example, GSM uses 1 bit per symbol. If the user file contains only 24 bits, enough for 24 symbols, the signal generator replicates the data pattern two more times to create a file with 72 bits. The expanded AUTOGEN_PRAM_1 file size would show 288 bytes (72 bits × 4 bytes/bit).

Use the following procedures to calculate the required amount of volatile memory for both framed and unframed TDMA signals:

- "Calculating Volatile Memory (PRAM) Usage for Unframed Data" on page 304
- "Calculating Volatile Memory (PRAM) Usage for Framed Data" on page 304

Calculating Volatile Memory (PRAM) Usage for Unframed Data

Use this procedure to calculate the memory size for either a bit or binary file. To properly demonstrate this process, the procedure employs a user file that contains 70 bytes (560 bits), with the bit file using only 557 bits.

1. Determine the AUTOGEN_PRAM_1 file size:

   The signal generator creates a 32–bit word for each user file bit (1 bit equals 4 bytes).

   Binary file 4 bytes × (70 bytes x 8 bits) = 2240 bytes

   Bit file 4 bytes × 557 bits= 2228 bytes

2. Calculate the number of memory blocks that the AUTOGEN_PRAM_1 file will occupy:

   Volatile memory allocates memory in blocks of 1024 bytes.

   Binary file 2240 / 1024 = 2.188 blocks

   Bit file 2228 / 1024 = 2.176 blocks

3. Round the memory block value to the next highest integer value.

   For this example, the AUTOGEN_PRAM_1 file will use three blocks of memory for a total of 3072 bytes.

4. Determine the number of memory blocks that the copy of the original file occupies in volatile memory.

   For this example the bit and binary file sizes are shown in the following list:

   - Binary file = 70 bytes < 1024 bytes = 1 memory block
   - Bit file = 80 bytes < 1024 bytes = 1 memory block

     Remember that a bit file includes a 10–byte file header.

5. Calculate the total volatile memory occupied by the user file data:

AUTOGEN_PRAM_1	Original File
3 blocks	1 block
1024 (3 + 1) = 4096 bytes	

Calculating Volatile Memory (PRAM) Usage for Framed Data

Framed data is not a selection for Custom, but it is for TDMA formats. To frame data, the signal generator adds framing overhead data such as tail bits, guard bits, and sync bits. These framing bits are in addition to the user file data. For more information on framed data, see “Understanding Framed Transmission For Real–Time TDMA” on page 315.

When using framed data, the signal generator views the data (framing and user file bits) in terms of the number of bits per frame, even if only one timeslot within a frame is active. This means that the signal generator creates a 32–bit word for each bit in a frame, for both active and inactive timeslots.

You can create a user file so that it fills a timeslot once or multiple times. When the user file fills a timeslot multiple times, the signal generator creates the same number of frames as the number of timeslots that the user file fills. For example, if a file contains enough data to fill a timeslot three times, the signal produces three new frames before the frames repeat. Each new frame increases the

AUTOGEN_PRAM_1 file size. If you select different user files for the timeslots within a frame, the user file that produces the largest number of frames determines the size of the AUTOGEN_PRAM_1 file.

Use this procedure to calculate the volatile memory usage for a GSM signal with two active timeslots and two user binary files. One user file, 57 bytes, is for a normal timeslot and another, 37 bytes, is for a custom timeslot.

1. Determine the total number of bits per timeslot.

   A GSM timeslot consists of 156.25 bits (control and payload data).

2. Calculate the number of bits per frame.

   A GSM frame consists of 8 timeslots: 8 × 156.25 = 1250 bits per frame

3. Determine how many bytes it takes to produce one frame in the signal generator:

   The signal generator creates a 32–bit word for each bit in the frame (1 bit equals 4 bytes).

   4 x 1250 = 5000 bytes

   Each GSM frame uses 5000 bytes of PRAM memory.

4. Analyze how many timeslots the user file data will fill.

   A normal GSM timeslot (TS) uses 114 payload data bits, and a custom timeslot uses 148 payload data bits. The user file (payload data) for the normal timeslot contains 57 bytes (456 bits) and the user file for the custom timeslot contains 37 bytes (296 bits).

   Normal TS 456 / 114 = 4 timeslots

   Custom TS 296 / 148 = 2 timeslots

NOTE Because there is an even number of bytes, either a bit or binary file works in this scenario. If there was an uneven number of bytes, a bit file would be the best choice to avoid data discontinuity.

---

5. Compute the number of frames that the signal generator will generate.

   There is enough user file data for four normal timeslots and two custom timeslots, so the signal generator will generate four frames of data.

6. Calculate the AUTOGEN_PRAM_1 file size:

Number of Frames	Bytes per Frame
4	5000
4 x 5000 = 20000 bytes	

7. Calculate the number of memory blocks that the AUTOGEN_PRAM_1 file will occupy:

   Volatile memory allocates memory in blocks of 1024 bytes.

   20000 / 1024 = 19.5 blocks

8. Round the memory block value up to the next highest integer value.

   For this example, the AUTOGEN_PRAM_1 file will use 20 blocks of memory for a total of 20480 bytes.

9. Determine the number of memory blocks that the original files occupy in volatile memory.

   The files do not share memory blocks, so you must determine how many memory blocks each file occupies.

Normal TS	Custom TS
57 bytes = 1 block	37 bytes = 1 block
1 + 1 = 2 memory blocks	

NOTE If the user file type is bit, remember to include the 10–byte file header in the file size.

10. Calculate the total volatile memory occupied by the AUTOGEN_PRAM_1 file and the user files:

AUTOGEN_PRAM_1	User Files
20 blocks	2 blocks
1024 (20 + 2) = 22528 bytes	

Downloading User Files

The signal generator expects bit and binary file type data to be downloaded as block data (binary data in bytes). The IEEE standard 488.2–1992 section 7.7.6 defines block data.

This section contains two examples to explain how to format the SCPI command for downloading user file data. The examples use the binary user file SCPI command, however the concept is the same for the bit file SCPI command:

- Command Format
- "Command Format in a Program Routine" on page 307

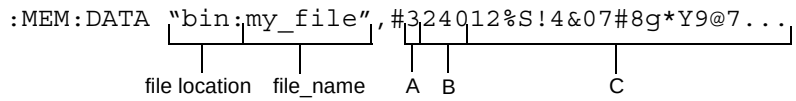
Command Format

This example conceptually describes how to format a data download command (`#ABC` represents the block data):

```
:MEM:DATA <"file_name">,#ABC
```

`<"file_name">`	the data file path and name
`#`	indicates the start of the block data
`A`	the number of decimal digits present in B
`B`	a decimal number specifying the number of data bytes to follow in C

C	the file data in bytes



bin:	the location of the file within the signal generator file system
my_file	the data file name as it will appear in the signal generator's memory catalog
#	indicates the start of the block data
3	B has three decimal digits
240	240 bytes (1,920 bits) of data to follow in C
12%S!4&07#8g*Y9@7...	the ASCII representation of some of the block data (binary data) downloaded to the signal generator, however not all ASCII values are printable

In actual use, the block data is not part of the command line as shown above, but instead resides in a binary file on the PC/UNIX. When the program executes the SCPI command, the command line notifies the signal generator that it is going to receive block data of the stated size and to place the file in the signal generator file directory with the indicated name. Immediately following the command execution, the program downloads the binary file to the signal generator. This is shown in the following section, "Command Format in a Program Routine"

Some commands are file location specific and do not require the file location as part of the file name. An example of this is the bit file SCPI command shown in "Command for Bit File Downloads" on page 309.

Command Format in a Program Routine

This section demonstrates the use of the download SCPI command within the confines of a C++ program routine. The following code sends the SCPI command and downloads user file data to the signal generator's Binary memory catalog (directory).

Line **Code–Download User File Data**

```
1  int bytesToSend;
2  bytesToSend = numsamples;
3  char s[20];
4  char cmd[200];
5  sprintf(s, "%d", bytesToSend);
6  sprintf(cmd, ":MEM:DATA \"BIN:FILE1\", #%d%d", strlen(s), bytesToSend);
7  iwrite(id, cmd, strlen(cmd), 0, 0);
8  iwrite(id, databuffer, bytesToSend, 0, 0);
9  iwrite(id, "\n", 1, 1, 0);
```

Line	Code Description–Download User File Data
1	Define an integer variable (*bytesToSend*) to store the number of bytes to send to the signal generator.
2	Calculate the total number of bytes, and store the value in the integer variable defined in line 1.
3	Create a string large enough to hold the *bytesToSend* value as characters. In this code, string *s* is set to 20 bytes (20 characters–one character equals one byte)
4	Create a string and set its length (*cmd*[200]) to hold the SCPI command syntax and parameters. In this code, we define the string length as 200 bytes (200 characters).
5	Store the value of *bytesToSend* in string *s*. For example, if bytesToSend = 2000; *s* = ”2000”. *sprintf()* is a standard function in C++, which writes string data to a string variable.
6	Store the SCPI command syntax and parameters in the string *cmd*. The SCPI command prepares the signal generator to accept the data. • *strlen()* is a standard function in C++, which returns length of a string. • If *bytesToSend* = 2000, then *s* = “2000”, *strlen(s)* = 4, so *cmd* = :MEM:DATA ”BIN:FILE1\” #42000.
7	Send the SCPI command stored in the string *cmd* to the signal generator contained in the variable *id*. • *iwrite()* is a SICL function in Agilent IO library, which writes the data (block data) specified in the string *cmd* to the signal generator. • The third argument of *iwrite()*, *strlen(cmd)*, informs the signal generator of the number of bytes in the command string. The signal generator parses the string to determine the number of data bytes it expects to receive. • The fourth argument of iwrite(), 0, means there is no END of file indicator for the string. This lets the session remain open, so the program can download the user file data.

Line	Code Description–Download User File Data
8	Send the user file data stored in the array *(databuffer)* to the signal generator. • *iwrite()* sends the data specified in *databuffer* to the signal generator (session identifier specified in *id*). • The third argument of *iwrite()*, *bytesToSend*, contains the length of the *databuffer* in bytes. In this example, it is 2000. • The fourth argument of *iwrite()*, 0, means there is no END of file indicator in the data. In many programming languages, there are two methods to send SCPI commands and data: — Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data. — Method 2 where the program sends a fixed number of bytes and ignores any zeros in the data. This is the method used in our program. For your programming language, you must find and use the equivalent of method two. Otherwise you may only achieve a partial download of the user file data.
9	Send the terminating carriage (\n) as the last byte of the waveform data. • *iwrite()* writes the data "\n" to the signal generator (session identifier specified in *id).* • The third argument of *iwrite()*, 1, sends one byte to the signal generator. • The fourth argument of *iwrite(),* 1, is the END of file indicator, which the program uses to terminate the data download. To verify the user file data download, see "Command for Bit File Downloads" on page 309 and "Commands for Binary File Downloads" on page 310.

Command for Bit File Downloads

Because the signal generator adds a 10–byte file header during a bit file download, you must use the SCPI command shown in Table 7-6. If you FTP or copy the file for the initial download, the signal generator does not add the 10–byte file header, and it does recognize the data in the file (no data in the transmitted signal).

Bit files enable you to control how many bits in the file the signal generator modulates onto the signal. Even with this file type, the signal generator requires that all data be contained within bytes. For more information on bit files, see "Bit File Type Data" on page 298.

Table 7-6 Bit File Type SCPI Commands

Type	Command Syntax
Command	`:MEM:DATA:BIT <"file_name">,<bit_count>,<block_data>` This downloads the file to the signal generator.

Table 7-6 Bit File Type SCPI Commands

Type	Command Syntax
Query	:MEM:DATA:BIT? <"file_name"> Within the context of a program this query extracts the user file data. Executing the query in a command window causes it to return the following information: <bit_count>,<block_data>.
Query	:MEM:CAT:BIT? This lists all of the files in the bit file directory and shows the remaining non–volatile memory: <bytes used by bit files>,<available non-volatile memory>,<"file_names">

Command Syntax Example

The following command downloads a file that contains 17 bytes:

```
:MEM:DATA:BIT "new_file",131,#21702%S!4&07#8g*Y9@7
```

Since this command is file specific (BIT), there is no need to add the file path to the file name.

After execution of this command, the signal generator creates a file in the bit directory (memory catalog) named “new_file” that contains 27 bytes. Remember that the signal generator adds a 10–byte file header to a bit file. When the signal generator uses this file, it will recognize only 131 of the 136 bits (17 bytes) contained in the file.

For information on downloading block data, see “Downloading User Files” on page 306.

Commands for Binary File Downloads

To download a user file as a binary file type means that the signal generator, when the file is selected for use, sees all of the data contained within the file. For more information on binary files, see “Binary File Type Data” on page 301. There are two ways to download the file: to be able to extract the file or not. Each method uses a different SCPI command, which is shown in Table 7-7.

Table 7-7 Binary File Type Commands

Command Type		Command Syntax
For Extraction	SCPI	:MEMory:DATA:UNPRotected "bin:file_name",<datablock> This downloads the file to the signal generator. You can extract the file within the context of a program.
	FTP[a]	put <file_name> /user/bin/file_name
No extraction		:MEM:DATA "bin:file_name",<block data> This downloads the file to the signal generator. You cannot extract the file.

Table 7-7 Binary File Type Commands

Command Type		Command Syntax
Query		:MEM:DATA? "bin:file_name" This returns information on the named file: <bit_count>,<block_data>. Within the context of a program, this query extracts the user file, provided it was download with the proper command.
Query		:MEM:CAT:BIN? This lists all of the files in the bit file directory and shows the remaining non–volatile memory: <bytes used by bit files>,<available non-volatile memory>,<"file_names">

a. See "FTP Procedures" on page 313.

File Name Syntax

There are three ways to format the file name, which must also include the file path:

- "BIN:file_name"
- "file_name@BIN"
- "/user/BIN/file_name"

Command Syntax Example

The following command downloads a file that contains 34 bytes:

```
:MEM:DATA "BIN:new_file",#2347^%S!4&07#8g*Y9@7.?:*Ru[+@y3#_^,>l
```

After execution of this command, the signal generator creates a file in the Binary (Bin) directory (memory catalog) named "new_file" that contains 34 bytes.

For information on downloading block data, see "Downloading User Files" on page 306.

Selecting a Downloaded User File as the Data Source

This section describes how to format SCPI commands for selecting a user file using commands from the GSM and Custom modulation formats. While the commands shown come from only two formats, the concept remains the same when making the data selection for any of the other real–time modulation formats that accept user data. To find the data selection commands for both framed and unframed data for the different modulation formats, see the signal generator's *SCPI Command Reference*.

1. For TDMA formats, select either framed or unframed data:

   ```
   :RADio:GSM:BURSt ON|OFF|1|0
   ```

   ON(1) = framed OFF(0) = unframed

2. Select the user file:

   Unframed Data

   ```
   :RADio:CUSTom:DATA "BIT:file_name"
   ```

   ```
   :RADio:CUSTom:DATA "BIN:file_name"
   ```

   Framed Data

   ```
   :RADio:GSM:SLOT0|1|2|3|4|5|6|7:NORMal:ENCRyption "BIT:file_name"
   ```

   ```
   :RADio:GSM:SLOT0|1|2|3|4|5|6|7:NORMal:ENCRyption "BIN:file_name"
   ```

3. Configure the remaining signal parameters.
4. Turn the modulation format on:

   ```
   :RADio:CUSTom:STATe On
   ```

Modulating and Activating the Carrier

Use the following commands to modulate the carrier and turn on the RF output. For a complete listing of SPCI commands, refer to the *SCPI Command Reference*.

```
:FREQuency:FIXed 2.5GHZ
```

```
:POWer:LEVel -10.0DBM
```

```
:OUTPut:MODulation:STATe ON
```

```
:OUTPut:STATe ON
```

Modifying User File Data

There are two ways to modify a file after downloading it to the signal generator:

- Use the signal generator's bit file editor. This works for both bit and binary files, but it converts a binary file to a bit file and adds a 10–byte file header. For more information on using the bit file editor, see the signal generator's *User's Guide*. You can also access the bit editor remotely using the signal generator's web server. For web server information, see the signal generator's *Programming Guide*.
- Use a hex editor program on your PC or UNIX workstation, as described below.

Modifying a Binary File with a Hex Editor

1. FTP the file to your PC/UNIX.

   For information on using FTP, see FTP Procedures. Ensure that you use binary file transfers during FTP operations.

2. Modify the file using a hex editor program.
3. FTP the file to the signal generator's BIN memory catalog (directory).

Modifying a Bit File with a Hex Editor

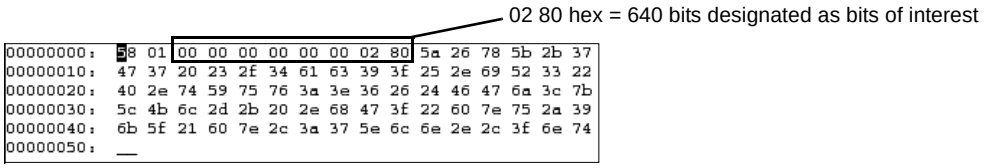
1. FTP the file to your PC/UNIX.

   For information on using FTP, see FTP Procedures. Ensure that you use binary file transfers during FTP operations.

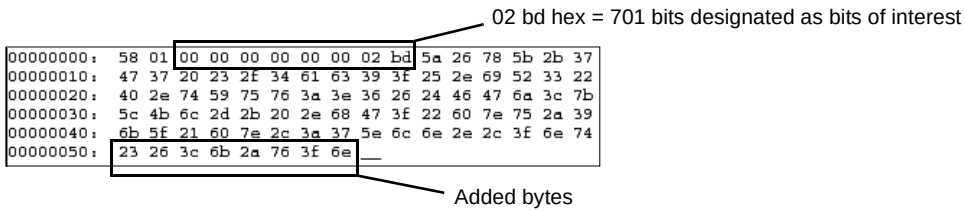
2. Modify the file using a hex editor program.

   If you need to decrease or increase the number of bits of interest, change the file header hex value.

80 Byte File From Signal Generator



Modified File (80 Bytes to 88 Bytes)



3. FTP the file to the signal generator's BIT memory catalog (directory).

FTP Procedures

CAUTION Avoid using the *OPC? or *WAI commands to verify that the FTP process has been completed. These commands can potentially hang up due to the processing of other SCPI parser operations. Refer to the *SCPI Command Reference*.

NOTE If you are remotely FTPing files and need to verify the completion of the FTP process, then query the instrument by using SCPI commands such as: ':MEM:DATA:', ':MEM:CAT', '*STB?', 'FREQ?', '*IDN?', 'OUTP:STAT?'. Refer to the *SCPI Command Reference*.

There are three ways to FTP a file:

- use Microsoft's ® Internet Explorer FTP feature
- use the signal generator's internal web server (ESG firmware ≥ C.03.76)

- use the PC or UNIX command window

Using Microsoft's Internet Explorer

1. Enter the signal generator's hostname or IP address as part of the FTP URL.

   ftp://<host name> or <IP address>
2. Press **Enter** on the keyboard or **Go** from the Internet Explorer window.

   The signal generator files appear in the Internet Explorer window.
3. Drag and drop files between the PC and the Internet Explorer window

---

Microsoft is a U.S registered trademark of Microsoft Corporation.

Using the Signal Generator's Internal Web Server

1. Enter the signal generator's hostname or IP address in the URL.

   http://<host name> or <IP address>

2. Click the **Signal Generator FTP Access** button located on the left side of the window.

   The signal generator files appear in the web browser's window.

3. Drag and drop files between the PC and the browser's window

For more information on the web server feature, see the *Programming Guide*.

Using the Command Window (PC or UNIX)

1. From the PC command prompt or UNIX command line, change to the proper directory:
   - When downloading from the signal generator, the directory in which to place the file.
   - When downloading to the signal generator, the directory that contains the file.
2. From the PC command prompt or UNIX command line, type `ftp <instrument name>`.

   Where `instrument name` is the signal generator's hostname or IP address.
3. At the `User:` prompt, press **Enter** (no entry is required).
4. At the `Password:` prompt, press **Enter** (no entry is required).
5. At the `ftp` prompt, type the desired command:

   To Get a File From the Signal Generator

   ```
   get /user/<directory>/<file_name1> <file_name>
   ```

   To Place a File in the Signal Generator

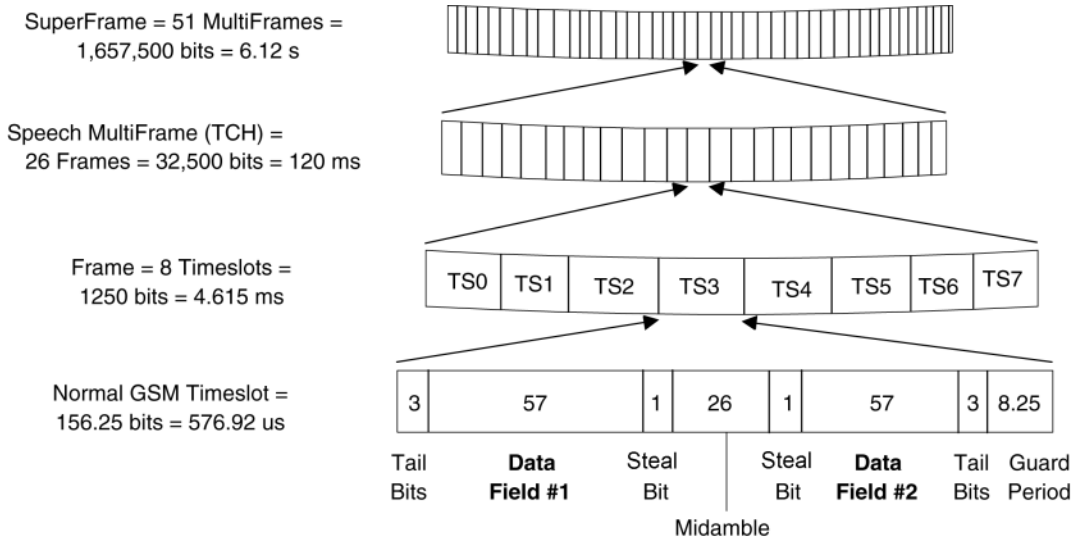
   ```
   put <file_name> /user/<directory>/<file_name1>
   ```

   - `<file_name1>` is the name of the file as it appears in the signal generator's directory.
   - `<file_name>` is the name of the file as it appears in the PC/UNIX current directory.
   - `<directory>` is the signal generator's `BIT` or `BIN` directory.
6. At the `ftp` prompt, type: `bye`
7. At the command prompt, type: `exit`

Understanding Framed Transmission For Real–Time TDMA

Specifying a user file as the data source for a framed transmission provides you with an easy method to multiplex real data into internally generated TDMA framing. The user file fills the data fields of the active timeslot in the first frame, and continue to fill the same timeslot of successive frames as long as there is more data in the file with enough bits to fill the data field. This functionality enables a communications system designer to download and modulate proprietary data sequences, specific PN sequences, or simulate multiframe transmission such as those specified by some mobile communications protocols. As the example in the following figure shows, a GSM multiframe transmission requires 26 frames for speech.

Figure 7-1 GSM Multiframe Transmission



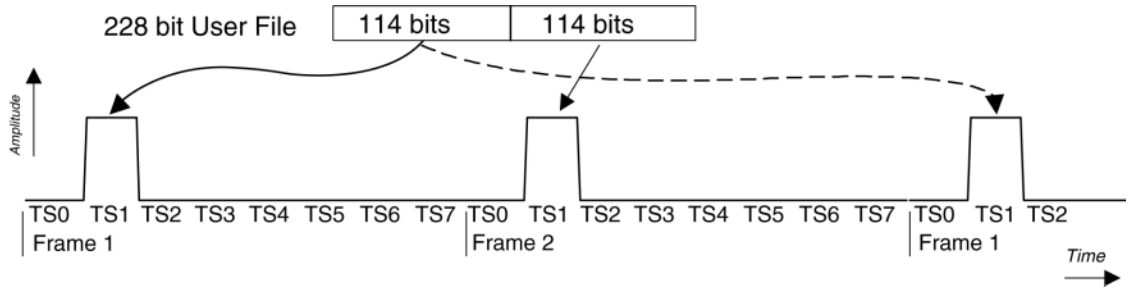
When you select a user file as the data source for a framed transmission, the signal generator's firmware loads PRAM with the framing protocol of the active TDMA format. This creates a file named AUTOGEN_PRAM_1 in addition to a copy of the user file. For all addresses corresponding to active (on) timeslots, the signal generator sets the burst bit to 1 and fills the data fields with the user file data. Other bits are set according to the configuration selected. For inactive (off) timeslots, the signal generator sets the burst control bit to 0, with the data being unspecified.

In the last byte that contains the last user file data bit, the signal generator sets the Pattern Reset bit to 1. This causes the user file data pattern to repeat in the next frame.

NOTE The data in PRAM is static. Firmware writes to PRAM once for the configuration selected and the hardware reads this data repeatedly. Firmware overwrites the volatile PRAM memory to reflect the desired configuration only when the data source or TDMA format changes.

For example, transmitting a 228–bit user file for timeslot #1 (TS1) in a normal GSM transmission creates two frames. Per the standard, a GSM normal channel is 156.25 bits long, with two 57–bit data fields (114 user data bits total per timeslot), and 42 bits for control or signalling purposes.The user file completely fills timeslot #1 for two consecutive frames, and then repeats. The seven remaining timeslots in the GSM frame are off, as shown in Figure 7-2

Figure 7-2 Mapping User File Data to a Single Timeslot



NOTE Compliant with the GSM standard, which specifies 156.25–bit timeslots, the signal generator uses 156–bit timeslots and adds an extra guard bit to every fourth timeslot.

For this protocol configuration, the signal generator's firmware loads PRAM with the bits defined in the following table. (These bits are part of the 32–bit word per frame bit.) The Pattern Reset bit, bit 7, is 0 for frame one and 1 for the last byte of frame two.

Frame	Timeslot	PRAM Word Offset	Data Bits	Burst Bits	Pattern Reset Bit
1	0	0 -155	0/1 (don't care)	0 (off)	0 (off)
1	1 (on)	156 - 311	set by GSM standard (42 bits) & first 114 bits of user file	1 (on)	0
1	2	312 - 467	0/1 (don't care)	0	0
1	3	468 - 624	0/1 (don't care)	0	0
1	4	625 - 780	0/1 (don't care)	0	0
1	5	781 - 936	0/1 (don't care)	0	0
1	6	937 - 1092	0/1 (don't care)	0	0
1	7	1093 - 1249	0/1 (don't care)	0	0
2	0	1250 - 1405	0/1 (don't care)	0	0
2	1 (on)	1406 - 1561	set by GSM standard (42 bits) & remaining bits of user file	1 (on)	0
2	2 through 6	1562 - 2342	0/1 (don't care)	0	0 (off)
2	7	2343 - 2499	0/1 (don't care)	0	1 (1 in offset 2499 only)

Event 1 output is set to 0 or 1 depending on the sync out selection, which enables the EVENT 1 output at either the beginning of the frame, beginning of a specific timeslot, or at all timeslots (SCPI command, `:RADio:GSM:SOUT FRAME|SLOT|ALL`).

Because timeslots are configured and enabled within the signal generator, a user file can be individually assigned to one or more timeslots. A timeslot cannot have more than one data source (PN sequence or user file) specified for it. The amount of user file data that can be mapped into hardware memory depends on both the amount of PRAM available on the baseband generator, and the number and size of each frame. (See "Determining Memory Usage for Custom and TDMA User File Data" on page 303.)

PRAM adds 31 bits to each bit in a frame, which forms 32–bit words.
The following shows how to calculate the amount of PRAM storage space required for a GSM superframe:

Bits per superframe = normal GSM timeslot × timeslot per frame × speech multiframe(TCH) × superframe

size of normal GSM timeslot = 156.25 bits timeslots per frame = 8 timeslots.

speech multiframe(TCH) = 26 frames superframe = 51 speech multiframes

1. Calculate the number of bits in the superframe:

   156.25 × 8 × 26 × 51 = 1,657,500 bits

2. Calculate the size of the PRAM file:

   1,657,500 bits × 4 bytes (32–bit words) = 6,630,000 bytes

3. Calculate how much memory the PRAM file will occupy

   6,630,000 bytes / 1,024 bytes per PRAM block = 6,474.6 memory blocks

4. Round the quotient up to the next integer value

   6,475 blocks × 1,024 bytes per block = 6,630,400 bytes

NOTE For the total PRAM memory usage, be sure to add the number of PRAM blocks that the user file occupies to the PRAM file size. For more information, see "Calculating Volatile Memory (PRAM) Usage for Framed Data" on page 304.

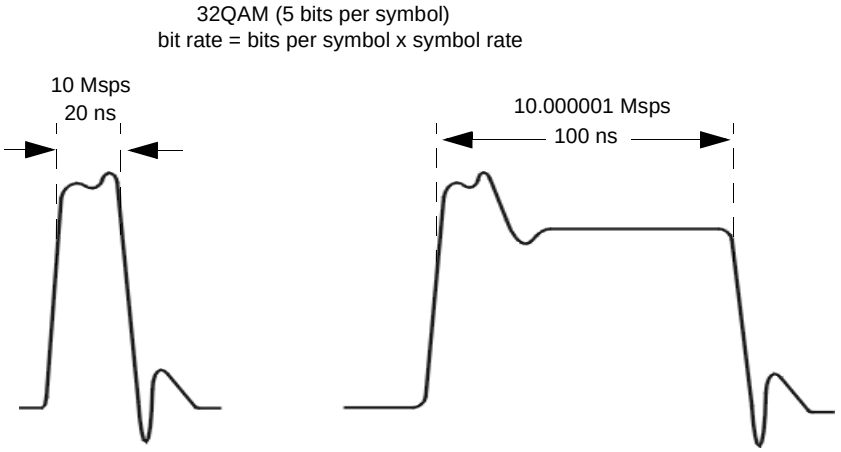
Real–Time Custom High Data Rates

Custom has two modes for processing data, serial and parallel. When the data bit–rate exceeds 50 Mbps, the signal generator processes data in parallel mode, which means processing the data symbol by symbol versus bit by bit (serial). This capability exists in only the Custom format when using a continuous data stream. This means that it does not apply to a downloaded PRAM file type (covered later in this chapter).

In parallel mode, for a 256QAM modulation scheme, Custom has the capability to reach a data rate of up to 400 Mbps. The FIR filter width is what determines the data rate. The following table shows the maximum data rate for each modulation type. Because the signal generator's maximum symbol rate is 50 Msps, a modulation scheme that has only 1 bit per symbol is always processed in serial mode.

Modulation Type	**Bit Rate Range for Internal Data (bit rate = symbol rate × bits per symbol)**		
	16 Symbol Wide FIR Filter	*32 Symbol Wide FIR Filter*	*64 Symbol Wide FIR Filter*
BPSK, 2FSK, MSK	1bps–50Mbps	1bps–25 Mbps	1bps–12.5Mbps
C4FM, OQPSK, 4FSK	2bps–100Mbps	2bps–50Mbps	2bps–25Mbps
IS95 OQPSK, QPSK			
P4DQPSK, IS95 QPSK			
GRAYQPSK, 4QAM			
D8PSK, EDGE, 8FSK, 8PSK	3bps–150Mbps	3bps–75Mbps	3bps–37.5Mbps
16FSK, 16PSK, 16QAM	4bps–200Mbps	4bps–100Mbps	4bps–50Mbps
Q32AM	5bps–250Mbps	5bps–125Mbps	5bps–62.5Mbps
64QAM	6bps–300Mbps	6bps–150Mbps	6bps–75Mbps
128QAM	7bps–350Mbps	7bps–175Mbps	7bps–87.5Mbps
256QAM	8bps–400Mbps	8bps–200Mbps	8bps–100Mbps

The only external effect of the parallel mode is in the EVENT 1 output signal. In serial and parallel mode, the signal generator outputs a narrow pulse at the EVENT 1 connector. But in parallel mode, the output pulse width increases by a factor of bits–per–symbol wide, as shown in the following figure.



NOTE: The pulse widths values are only for example purposes. The actual width may vary from the above values.

Pattern RAM (PRAM) Data Downloads (E4438C and E8267D)

NOTE This section applies only to the E4438C with Option 001, 002, 601, or 602, and the E8267D with Option 601 or 602.

If you encounter problems with this section, refer to "Data Transfer Troubleshooting (N5162A, N5182A, E4438C and E8267D Only)" on page 351.

To verify the SCPI parser's responsiveness when remotely using the :MEM:DATA SCPI command to upload files, the file's upload should be verified using the *STB? command. Refer to the *SCPI Command Reference*.

This section contains information to help you transfer user–generated PRAM data from a system controller to the signal generator's PRAM. It explains how to download data directly into PRAM and modulate the carrier signal with the data.

The control bits included in the PRAM file download, control the following signal functions:

- bursting
- timing signal at the EVENT 1 rear panel connector
- data pattern reset

PRAM data downloads apply to only real–time Custom and TDMA modulation formats. In the TDMA formats, PRAM files are available only while using the unframed data selection. The following table on page 321 shows which signal generator models support these formats.

E4438C ESG		E2867D PSG
Custom[a]	TDMA[b]	Custom[a]

a. For ESG, requires Option 001, 002, 601, or 602, for PSG requires Option 601 or 602.
b. Real–time TDMA modulation formats require Option 402 and include EDGE, GSM, NADC, PDC, PHS, DECT, and TETRA.

PRAM files differ from bit and binary user files.

Bit and binary user files (see page 297) download to non–volatile memory and the signal generator loads the user file data into PRAM (volatile/waveform memory) for use. The signal generator adds the required control bits when it generates the signal.

A PRAM file downloads directly into PRAM, and it includes seven of the required control bits for each data (payload) bit. The signal generator adds the remaining control bits when it generates the signal. You download the file using either a list or block data format. Programs such as MATLAB or MathCad can generate the data.

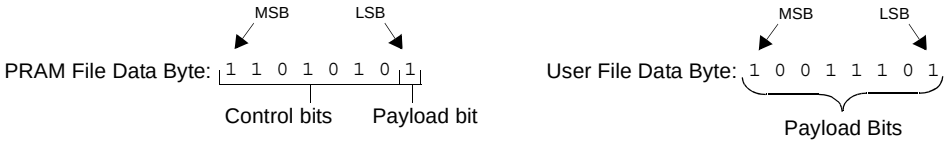
This type of signal control enables you to design experimental or proprietary framing schemes.

After selecting the PRAM file, the signal generator builds the modulation scheme by reading data stored in PRAM, and constructing framing protocols according to the PRAM file data and the modulation format. You can manipulate PRAM data by changing the standard protocols for a modulation format such as the symbol rate, modulation type, and filter either through the front panel interface or with SCPI commands.

Understanding PRAM Files

The term PRAM file comes from earlier Agilent products, the E443xB ESGs. PRAM is another term for waveform memory (WFM1), which is also known as volatile memory. This means that PRAM files and waveform files occupy the same memory location. The signal generator's volatile memory (waveform memory) storage path is `/user/BBG1/waveform`. For more information on memory, see "Signal Generator Memory" on page 291.

The following figure shows a PRAM byte and illustrates the difference between it and a bit/binary user file byte. Notice the control bits in the PRAM byte.



Only three of the seven control bits elicit a response from the signal generator. The other four bits are reserved. Table 7-8 describes the bits for a PRAM byte.

Table 7-8 PRAM Data Byte

Bit	Function	Value	Comments
0	Data	0/1	This is the data (payload) bit. It is "unspecified" when burst (bit 2) is set to 0.

Table 7-8 PRAM Data Byte

Bit	Function	Value	Comments
1	Reserved	0	Always 0
2	Burst	0/1	1 = RF on 0 = RF off For non–bursted, non–TDMA systems, to have a continuous signal, set this bit to 1 for all bytes. For framed data, set this bit to 1 for *on* timeslots and 0 for *off* timeslots.
3	Reserved	0	Always 0
4	Reserved	1	Always 1
5	Reserved	0	Always 0
6	EVENT1 Output	0/1	To have the signal generator output a single pulse at the EVENT 1 connector, set this bit to 1. Use this output for functions such as a triggering external hardware to indicate when the data pattern begins and restarts, or creating a data–synchronous pulse train by toggling this bit in alternate bytes.
7	Pattern Reset	0/1	0 = continue to next sequential memory address. 1 = end of memory and restart memory playback. This bit is set to 0 for all bytes except the last byte of PRAM. To restart the pattern, set the last byte of PRAM to 1.

As seen in Table 7-8, only four bits, shown in the following list, can change state:

- bit 0—data
- bit 2—bursting
- bit 6—EVENT 1 rear panel output
- bit 7—pattern reset

Because a PRAM byte has only four bits that can change states, there are only 15 possible byte patterns as shown in Table 7-9. The table also shows the decimal value for each pattern, which is needed for downloading data using the list format shown on page 325.

Table 7-9 PRAM Byte Patterns and Bit Positions

Bit Function	Pattern Reset	EVENT 1 Output	Reserved (Bit = 0)	Reserved (Bit = 1)	Reserved (Bit = 0)	Burst	Reserved (Bit = 0)	Data	Bit Pattern Decimal Value
Bit Position	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	---
Bit Pattern	1	1	0	1	0	1	0	1	213
	1	1	0	1	0	1	0	0	212
	1	1	0	1	0	0	0	1	209
	1	1	0	1	0	0	0	0	208
	1	0	0	1	0	1	0	1	149
	1	0	0	1	0	0	0	1	145
	1	0	0	1	0	0	0	0	144
	0	1	0	1	0	1	0	1	85
	0	1	0	1	0	1	0	0	84
	0	1	0	1	0	0	0	1	81
	0	1	0	1	0	0	0	0	80
	0	0	0	1	0	1	0	1	21
	0	0	0	1	0	1	0	0	20
	0	0	0	1	0	0	0	1	17
	0	0	0	1	0	0	0	0	16

Viewing the PRAM Waveform

After the waveform data is written to PRAM, the data pattern can be viewed using an oscilloscope. There is approximately a 12–symbol delay between a state change in the burst bit and the corresponding effect at the RF out. This delay varies with symbol rate and filter settings, and requires compensation to advance the burst bit in the downloaded PRAM file.

PRAM File Size

Because volatile memory resides on the baseband generator (BBG), the maximum PRAM file size depends on the installed baseband generator option, as shown in Table 7-10.

Table 7-10 Maximum PRAM User File Size (Payload Bits Only)

Modulation Format	Baseband Generator Option		
	001, 601	002	602
Custom TDMA	8 Mbits[a]	32 Mbits[a]	64 Mbits[a]

a. File size with no other files residing in volatile memory.

The maximum PRAM user file size in the table above refers to the maximum number of payload bits. After downloading, the signal generator translates each downloaded payload bit into a 32–bit word:

- 1 downloaded payload bit
- 7 downloaded control bits as shown in Table 7-8 on page 321
- 24 bits added by the signal generator

The following table shows the maximum file size after the signal generator has translated the maximum number of payload bits into 32–bit words.

Table 7-11 Maximum File Size After Downloading

Modulation Format	Baseband Generator Option		
	001, 601	002	602
Custom TDMA	32 MBytes[a]	128 MBytes[a]	256 MBytes[a]

a. File size with no other files residing in volatile memory.

To properly size a PRAM file, you must determine the file size after the 32–bit translation process. The signal generator measures a PRAM file size in units of bytes; each 32–bit word equals 4 bytes.

Determining the File Size

The following example shows how to calculate a downloaded file size using a PRAM file that contains 89 bytes (payload bits plus 7 control bits per payload bit):

89 bytes + [(89 × 24 bits) / 8] = 356 bytes

Because the file downloads one fourth of the translated 32–bit word, another method to calculate the file size is to multiply the downloaded file size by four:

89 bytes × 4 = 356 bytes

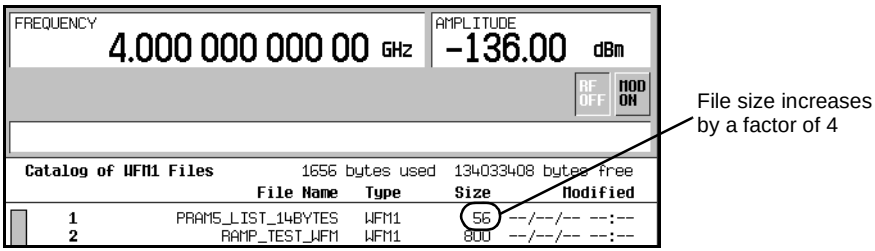
See also "Signal Generator Memory" on page 291 and "Checking Available Memory" on page 295.

Minimum File Size

A PRAM file requires a minimum of 60 bytes to create a signal. If the downloaded file contains less than 60 bytes, the signal generator replicates the file until the file size meets the 60 byte minimum. This replication process occurs after you select the file and turn the modulation format on. The following example shows this process using a downloaded 14–byte file:

- During the file download, the 14 bytes are translated into 56 bytes (fourteen 32–bit words).

  14 bytes × 4 = 56 bytes



- After selecting and turning the format on, the signal generator replicates the file contents to create the 60 byte minimum file size

  60 bytes / 14 bytes = 4.29 file replications

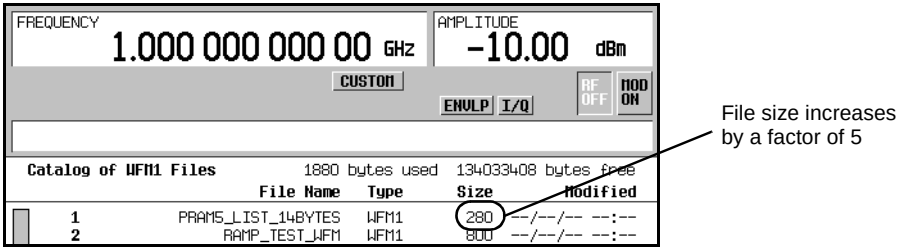
  The signal generator rounds this real value up to the next highest integer. In this example, the signal generator replicates the fourteen 32–bit words (56 bytes) by a factor of 5, which makes the final file size 280 bytes. This equates to a 70 byte file.

  14 bytes × 5 = 70 bytes

  70 + [(70 × 24) / 8] = 280 bytes

  Or

  56 bytes × 5 = 280 bytes



SCPI Command for a List Format Download

Using the list format, enter the data in the command line using comma separated decimal values. This file type takes longer to download because the signal generator must parse the data. When creating the data, remember that the signal generator requires a minimum of 60 bytes. For more information on file size limits, see “PRAM File Size” on page 324.

Command Syntax

```
:MEMory:DATA:PRAM:FILE:LIST <"file_name">,<uint8>[,<uint8>,<...>]
```

`uint8` The decimal equivalent of an unsigned 8–bit integer value. For a list of usable decimal values and their meaning with respect to the generated signal, see Table 7-9 on page 323.

Command Syntax Example

The following example, when executed, creates a new file in volatile (waveform) memory with the following attributes:

- creates a file named *new_file*
- outputs a single pulse at the EVENT 1 connector
- bursts the data pattern 1100 seven times over 28 bytes
- transmits 32 non–bursted bytes
- resets the data pattern so it starts again

```
:MEMory:DATA:PRAM:FILE:LIST <"new_file">,85,21,20,20,21,21,20,20,21,21,20,20,21,21,
20,20,21,21,20,20,21,21,20,20,21,21,20,20,16,16,16,16,16,16,16,16,16,16,16,16,16,
16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,144
```

The following list defines the meaning of the different bytes seen in the command line:

85 Send a pulse to the EVENT 1 output, and burst the signal with a data bit of 1.

21 Burst the signal with a data bit of 1.

20 Burst the signal with a data bit of 0.

16 Do not burst the signal (RF output off), and set the data bit to 0.

144 Reset the data pattern, do not burst the signal (RF output off), and set the data bit to 0.

SCPI Command for a Block Data Download

The IEEE standard 488.2–1992 section 7.7.6 defines block data. The signal generator is able to download block data significantly faster than list formatted data (see page 325), because it does not have to parse the data. When creating the data, remember that the signal generator requires a minimum of 60 bytes. For more information on file size limits, see "PRAM File Size" on page 324.

Command Syntax

```
:MEMory:DATA:PRAM:FILE:BLOCk <"file_name">,<blockdata>
```

The following sections explain how to format the SCPI command for downloading block data:

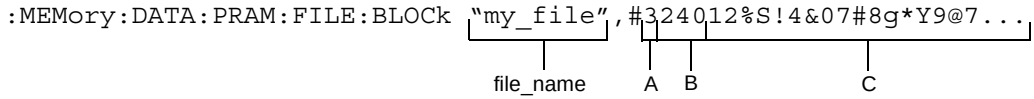
- Command Syntax Example
- Command Syntax in a Program Routine

Command Syntax Example

This example conceptually describes how to format a block data download command (#ABC represents the block data):

```
:MEMory:DATA:PRAM:FILE:BLOCk <"file_name">,#ABC
```

<"file_name">	the file name as it will appear in the signal generator
#	indicates the start of the block data
A	the number of decimal digits present in B
B	a decimal number specifying the number of data bytes to follow in C
C	the PRAM file data in bytes



my_file	the PRAM file name as it will appear in the signal generator's WFM1 memory catalog
#	indicates the start of the block data
3	B has three decimal digits
240	240 bytes of data to follow in C
12%S!4&07#8g*Y9@7...	the ASCII representation of some of the block data (binary data) downloaded to the signal generator, however not all ASCII values are printable

In actual use, the block data is not part of the command line as shown above, but instead resides in a binary file on the PC/UNIX. When the program executes the SCPI command, the command line notifies the signal generator that it is going to receive block data of the stated size, and to place the file in the signal generator file directory with the indicated name. Immediately following the command execution, the program downloads the binary file to the signal generator. This is shown in the following section, "Command Syntax in a Program Routine"

Command Syntax in a Program Routine

This section demonstrates the use of the download SPCI command within the confines of a C++ program routine. The following code sends the SCPI command and downloads a 240 byte PRAM file to the signal generator's WFM1 (waveform) memory catalog. This program assumes that there is a char array, *databuffer*, that contains the 240 bytes of PRAM data and that the variable *numbytes* stores the length of the array.

Line **Code–Download PRAM File Data**

```
1  int bytesToSend;
2  bytesToSend = numbytes;
3  char s[4];
4  char cmd[200];
5  sprintf(s, "%d", bytesToSend);
6  sprintf(cmd, ":MEM:DATA:PRAM:FILE:BLOCk \"FILE1\", #%d%d", strlen(s),bytesToSend);
7  iwrite(id, cmd, strlen(cmd), 0, 0);
8  iwrite(id, databuffer, bytesToSend, 0, 0);
9  iwrite(id, "\n", 1, 1, 0);
```

Line	Code Description–Download PRAM File Data
1	Define an integer variable (*bytesToSend*) to store the number of bytes to send to the signal generator.
2	Store the total number of PRAM bytes in the integer variable defined in line 1. *numbytes* contains the length of the *databuffer* array referenced in line 8.
3	Create a string large enough to hold the *bytesToSend* value as characters plus a null character value. In this code, string *s* is set to 4 bytes (3 characters for the bytesToSend value and one null character–one character equals one byte).
4	Create a string and set its length (*cmd*[200]) to hold the SCPI command syntax and parameters. In this code, we define the string length as 200 bytes (200 characters).
5	Store the value of *bytesToSend* in string *s*. For this example, bytesToSend = 240; *s* = ”240”
6	Store the SCPI command syntax and parameters in the string *cmd*. The SCPI command prepares the signal generator to accept the data. • *sprintf()* is a standard function in C++, which writes string data to a string variable. • *strlen()* is a standard function in C++, which returns length of a string. • *bytesToSend* = 240, then *s* = “240” plus the null character, *strlen(s)* = 4, so *cmd* = :MEM:DATA:PRAM:FILE:BLOCk ”FILE1\” #3240.
7	Send the SCPI command stored in the string *cmd* to the signal generator contained in the variable *id*. • *iwrite()* is a SICL function in Agilent IO library, which writes the data (block data) specified in the string *cmd* to the signal generator. • The third argument of *iwrite()*, *strlen(cmd)*, informs the signal generator of the number of bytes in the command string. The signal generator parses the string to determine the number of data bytes it expects to receive. • The fourth argument of iwrite(), 0, means there is no END of file indicator for the string. This lets the session remain open, so the program can download the PRAM file data.

Line	Code Description–Download PRAM File Data
8	Send the PRAM file data stored in the array, *databuffer*, to the signal generator. • *iwrite()* sends the data specified in *databuffer* (PRAM data) to the signal generator (session identifier specified in *id*). • The third argument of *iwrite()*, *bytesToSend*, contains the length of the *databuffer* in bytes. In this example, it is 240. • The fourth argument of *iwrite()*, 0, means there is no END of file indicator in the data. In many programming languages, there are two methods to send SCPI commands and data: — Method 1 where the program stops the data download when it encounters the first zero (END indicator) in the data. — Method 2 where the program sends a fixed number of bytes and ignores any zeros in the data. This is the method used in our program. For your programming language, you must find and use the equivalent of method two. Otherwise you may only achieve a partial download of the user file data.
9	Send the terminating carriage (\n) as the last byte of the waveform data. • *iwrite()* writes the data "\n" to the signal generator (session identifier specified in *id).* • The third argument of *iwrite()*, 1, sends one byte to the signal generator. • The fourth argument of *iwrite(),* 1, is the END of file indicator, which the program uses to terminate the data download.

Selecting a Downloaded PRAM File as the Data Source

The following steps show the process for selecting a PRAM file using commands from the GSM (TDMA) modulation format. While the commands shown come from only one format, the concept remains the same when making the data selection for any of the other real–time modulation formats that support PRAM data. To find the commands for Custom and the other TDMA formats, see the signal generator's *SCPI Command Reference*.

1. For real–time TDMA formats, select unframed data:

   ```
   :RADio:GSM:BURSt:STATe OFF
   ```

2. Select the data type:

   ```
   :RADio:GSM:DATA PRAM
   ```

3. Select the PRAM file:

   ```
   :RADio:GSM:DATA:PRAM <"file_name">
   ```

   Because the command is file specific (PRAM), there is no need to include the file path with the file name.

4. Configure the remaining signal parameters.

5. Turn the modulation format on:

```
:RADio:GSM:STATe On
```

Modulating and Activating the Carrier

Use the following commands to modulate the carrier and turn on the RF output. For a complete listing of SPCI commands, refer to the *SCPI Command Reference*.

```
:FREQuency:FIXed 1.8GHZ
:POWer:LEVel -10.0DBM
:OUTPut:MODulation:STATe ON
:OUTPut:STATe ON
```

Storing a PRAM File to Non–Volatile Memory and Restoring to Volatile Memory

After you download the file to volatile memory (waveform memory), you can then save it to non–volatile memory. Remember that a PRAM file downloads to waveform memory. Conversely, when you store a PRAM file to non–volatile memory, it uses the same directory as waveform files. When storing or restoring a file, you must include the file path as part of the file_name variable.

Command Syntax

The first file_name variable is the current location of the file and its name; the second file_name variable is the destination to store the file and its name.

There are three ways to format the file_name variable to include the file path:

Volatile Memory to Non–Volatile Memory

```
:MEMory:COPY "WFM1:file_name","NVWFM:file_name"
:MEMory:COPY "file_name@WFM1","file_name@NVWFM"
:MEMory:COPY "/user/bbg1/waveform/file_name","/user/waveform/file_name"
```

Non–Volatile Memory to Volatile Memory

```
:MEMory:COPY "NVWFM:file_name","WFM1:file_name"
:MEMory:COPY "file_name@NVWFM","file_name@WFM1"
:MEMory:COPY "/user/waveform/file_name","/user/bbg1/waveform/file_name"
```

Extracting a PRAM File

When you extract a PRAM file, you are extracting the translated 32–bit word–per–byte file. You cannot extract just the downloaded data. Extracting a PRAM file is similar to extracting a waveform file in that you use the same commands, and the PRAM file resides in either volatile memory (waveform memory) or the waveform directory for non–volatile memory. After extraction, you can download the file to the same signal generator or to another signal generator with the proper option configuration that supports the downloaded file. There are two ways to download a file after extraction:

- with the ability to extract later

- with no extraction capability

CAUTION Ensure that you do not use the `:MEMory:DATA:PRAM:FILE:BLOCk` command to download an extracted file. If you use this command, the signal generator will treat the file as a new PRAM file and translate the LSB of each byte into a 32–bit word, corrupting the file data.

Command Syntax

This section lists the commands for extracting PRAM files and downloading extracted PRAM files. To download an extracted file, you must use block data. For information on block data, see "SCPI Command for a Block Data Download" on page 326. In addition, there are three ways to format the file_name variable, which must also include the file path, as shown in the following tables.

There are two commands for file extraction:

- `:MEM:DATA? <"file_name">`
- `:MMEM:DATA? <"filename">`

The following table uses the first command to illustrate the command format, however the format is the same if you use the second command.

Table 7-12 Extracting a PRAM File

Extraction Method/Memory Type	Command Syntax Options

SCPI/volatile memory	`:MEM:DATA? "WFM1:file_name"`
	`:MEM:DATA? "file_name@WFM1"`
	`:MEM:DATA? "/user/bbg1/waveform/file_name"`

SCPI/non–volatile memory	`:MEM:DATA? "NVWFM:file_name"`
	`:MEM:DATA? "file_name@NVWFM"`
	`:MEM:DATA? "/user/waveform/file_name"`

FTP/volatile memory[a]	`get /user/bbg1/waveform/file_name`
FTP/non–volatile memory[a]	`get /user/waveform/file_name`

a. See "FTP Procedures" on page 313.

Table 7-13 Downloading a File for Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory	:MEM:DATA:UNPRotected "WFM1:file_name",<blockdata> :MEM:DATA:UNPRotected "file_name@WFM1",<blockdata> :MEM:DATA:UNPRotected "/user/bbg1/waveform/file_name",<blockdata>
SCPI/non–volatile memory	:MEM:DATA:UNPRotected "NVWFM:file_name",<blockdata> :MEM:DATA:UNPRotected "file_name@NVWFM",<blockdata> :MEM:DATA:UNPRotected "/user/waveform/file_name",<blockdata>
FTP/volatile memory[a]	put <file_name> /user/bbg1/waveform/file_name
FTP/non–volatile memory[a]	put <file_name> /user/waveform/file_name

a. See "FTP Procedures" on page 313.

There are two commands that download a file for no extraction:

- :MEM:DATA <"file_name">,<blockdata>
- :MMEM:DATA <"filename">,<blockdata>

The following table uses the first command to illustrate the command format, however the format is the same if you use the second command.

Table 7-14 Downloading a File for No Extraction

Download Method/ Memory Type	Command Syntax Options
SCPI/volatile memory	:MEM:DATA "WFM1:file_name",<blockdata> :MEM:DATA "file_name@WFM1",<blockdata> :MMEM:DATA "user/bbg1/waveform/file_name",<blockdata>
SCPI/non–volatile memory	:MEM:DATA "NVWFM:file_name",<blockdata> :MEM:DATA "file_name@NVWFM",<blockdata> :MEM:DATA /user/waveform/file_name",<blockdata>

Modifying PRAM Files

The only way to change PRAM file data is to modify the original file on a computer and download it again. The signal generator does not support viewing and editing PRAM file contents. Because the signal generator translates the data bit into a 32–bit word, the file contents are not recognizable, and therefore not editable using a hex editor program, as shown in the following figure.

60 byte PRAM file prior to downloading

```
00000000:  35 15 15 15 15 14 15 14 15 14 14 15 15 15 14 14
00000010:  14 15 15 15 14 15 14 14 15 14 14 15 15 14 14 15
00000020:  15 14 14 14 14 14 15 15 14 14 15 15 14 15 15 14
00000030:  14 14 15 14 14 15 15 15 15 15 14 90 __
```

60 byte PRAM file after downloading

```
00000000:  00 01 01 40 00 01 00 40 00 01 00 40 00 01 00 40
00000010:  00 01 00 40 00 00 00 40 00 01 00 40 00 00 00 40
00000020:  00 01 00 40 00 00 00 40 00 00 00 40 00 01 00 40
00000030:  00 01 00 40 00 01 00 40 00 00 00 40 00 00 00 40
00000040:  00 00 00 40 00 01 00 40 00 01 00 40 00 01 00 40
00000050:  00 00 00 40 00 01 00 40 00 00 00 40 00 00 00 40
00000060:  00 01 00 40 00 00 00 40 00 00 00 40 00 01 00 40
00000070:  00 01 00 40 00 00 00 40 00 00 00 40 00 01 00 40
00000080:  00 01 00 40 00 00 00 40 00 00 00 40 00 00 00 40
00000090:  00 00 00 40 00 00 00 40 00 01 00 40 00 01 00 40
000000a0:  00 00 00 40 00 00 00 40 00 01 00 40 00 01 00 40
000000b0:  00 00 00 40 00 01 00 40 00 01 00 40 00 00 00 40
000000c0:  00 00 00 40 00 00 00 40 00 01 00 40 00 00 00 40
000000d0:  00 00 00 40 00 01 00 40 00 01 00 40 00 01 00 40
000000e0:  00 01 00 40 00 01 00 40 00 00 00 40 00 00 00 00
000000f0:  __
```

FIR Filter Coefficient Downloads (N5162A, N5182A, E4438C and E8267D)

NOTE If you encounter problems with this section, refer to "Data Transfer Troubleshooting (N5162A, N5182A, E4438C and E8267D Only)" on page 351.

The signal generator accepts finite impulse response (FIR) filter coefficient downloads. After downloading the coefficients, these user–defined FIR filter coefficient values can be selected as the filtering mechanism for the active digital communications standard.

Data Requirements

There are two requirements for user–defined FIR filter coefficient files:

1. Data must be in ASCII format.

   The signal generator processes FIR filter coefficients as floating point numbers.

2. Data must be in List format.

   FIR filter coefficient data is processed as a list by the signal generator's firmware. See Sample Command Line.

Data Limitations

Filter lengths of up to 1024 taps (coefficients) are allowed. The oversample ratio (OSR) is the number of filter taps per symbol. Oversample ratios from 1 through 32 are possible.

The maximum combination of OSR and symbols allowed is 32 symbols with an OSR of 32.

The Real Time I/Q Baseband FIR filter files are limited to 1024 taps, 64 symbols and a 16–times oversample ratio. FIR filter files with more than 64 symbols cannot be used.

The ARB Waveform Generator FIR filter files are limited to 512 taps and 512 symbols.

The sampling period (Δt) is equal to the inverse of the sampling rate (FS). The sampling rate is equal to the symbol rate multiplied by the oversample ratio. For example, the GSM symbol rate is 270.83 ksps. With an oversample ratio of 4, the sampling rate is 1083.32 kHz and Δt (inverse of FS) is 923.088 nsec.

Downloading FIR Filter Coefficient Data

The signal generator stores the FIR files in the FIR (/USER/FIR) directory, which utilizes non–volatile memory (see also "Signal Generator Memory" on page 291). Use the following SCPI command line to download FIR filter coefficients (file) from the PC to the signal generator's FIR directory:

```
:MEMory:DATA:FIR <"file_name">,osr,coefficient{,coefficient}
```

Use the following SCPI command line to query list data from the FIR file:

```
:MEMory:DATA:FIR? <"file_name">
```

Sample Command Line

The following SCPI command will download a typical set of FIR filter coefficient values and name the file "FIR1":

```
:MEMory:DATA:FIR "FIR1",4,0,0,0,0,0,0.000001,0.000012,0.000132,0.001101,
0.006743,0.030588,0.103676,0.265790,0.523849,0.809508,1,1,0.809508,0.523849,
0.265790,0.103676,0.030588,0.006743,0.001101,0.000132,0.000012,0.000001,0,
0,0,0,0
```

`FIR1`	assigns the name FIR1 to the associated OSR (over sample ratio) and coefficient values (the file is then represented with this name in the FIR File catalog)
`4`	specifies the oversample ratio
`0,0,0,0,0,` `0.000001,...`	the FIR filter coefficients

Selecting a Downloaded User FIR Filter as the Active Filter

NOTE For information on manual key presses for the following remote procedures, refer to the *User's Guide*.

FIR Filter Data for TDMA Format

The following remote command selects user FIR filter data as the active filter for a TDMA modulation format.

```
:RADio:<desired format>:FILTer <"file_name">
```

This command selects the user FIR filter, specified by the file name, as the active filter for the TDMA modulation format. After selecting the file, activate the TDMA format with the following command:

```
:RADio:<desired format>:STATe On
```

FIR Filter Data for Custom Modulation

The following remote command selects user FIR filter data as the active filter for a custom modulation format.

```
:RADio:CUSTom:FILTer <"file_name">
```

This command selects the user FIR filter, specified by the file name, as the active filter for the custom modulation format. After selecting the file, activate the TDMA format with the following command:

```
:RADio:CUSTom:STATe On
```

FIR Filter Data for CDMA and W–CDMA Modulation

The following remote command selects user FIR filter data as the active filter for a CDMA modulation format. The process is very similar for W–CDMA.

```
:RADio:<desired format>:ARB:FILTer <"file_name">
```

This command selects the User FIR filter, specified by the file name, as the active filter for the CDMA or W–CDMA modulation format. After selecting the file, activate the CDMA or W–CDMA format with the following command:

```
:RADio:<desired format>:ARB:STATe On
```

Modulating and Activating the Carrier

The following commands set the carrier frequency and power, and turns on the modulation and the RF output.

1. Set the carrier frequency to 2.5 GHz:

   ```
   :FREQuency:FIXed 2.5GHZ
   ```

2. Set the carrier power to –10.0 dBm:

   ```
   :POWer:LEVel -10.0DBM
   ```

3. Activate the modulation:

   ```
   :OUTPut:MODulation:STATe ON
   ```

4. Activate the RF output:

   ```
   :OUTPut:STATe ON
   ```

Save and Recall Instrument State Files

NOTE References to waveform files and some of the other data file types mentioned in the following sections are not available for all models and options of signal generator. Refer to the instrument's *Data Sheet* for the signal generator and options being used.

The signal generator can save instrument state settings to memory. An instrument state setting includes any instrument state that does not survive a signal generator preset or power cycle such as frequency, amplitude, attenuation, and other user–defined parameters. The instrument state settings are saved in memory and organized into sequences and registers. There are 10 sequences with 100 registers per sequence available for instrument state settings. These instrument state files are stored in the USER/STATE directory. See also, "Signal Generator Memory" on page 291.

The save function does not store data such as Arb waveforms, table entries, list sweep data, and so forth. The save function saves a reference to the waveform or data file name associated with the instrument state. Use the store commands or store softkey functions to store these data file types to the signal generator's memory catalog.

Before saving an instrument state that has a data file or waveform file associated with it, store the file. For example, if you are editing a multitone arb format, store the multitone data to a file in the signal generator's memory catalog (multitone files are stored in the USER/MTONE directory). Then save the instrument state associated with that data file. The settings for the signal generator such as frequency and amplitude and a reference to the multitone file name will be saved in the selected sequence and register number. Refer to the signal generator's *User's Guide*, *Key and Data Field Reference*, or the signal generator's Help hardkey for more information on the save and recall functions.

Save and Recall SCPI Commands

The following command sequence saves the current instrument state, using the *SAV command, in register 01, sequence 1. A comment is then added to the instrument state.

```
*SAV 01,1
:MEM:STAT:COMM 01,1,"Instrument state comment"
```

If there is a waveform or data file associated with the instrument state, there will be a file name reference saved along with the instrument state. However, the waveform/data file must be stored in the signal generator's memory catalog as the *SAV command does not save data files. For more information on storing file data such as modulation formats, arb setups, and table entries refer to the signal generator's *User's Guide*.

NOTE On the N5162A, N5182A, E4438C, and E8267D, if a saved instrument state contains a reference to a waveform file, ensure that the waveform file resides in volatile memory before recalling the instrument state. For more information, see the *User's Guide*.

The recall function recalls a saved instrument state. If there is a data file associated with the instrument state, the file will be loaded along with the instrument state. The following command recalls the instrument state saved in register 01, sequence 1.

```
*RCL 01,1
```

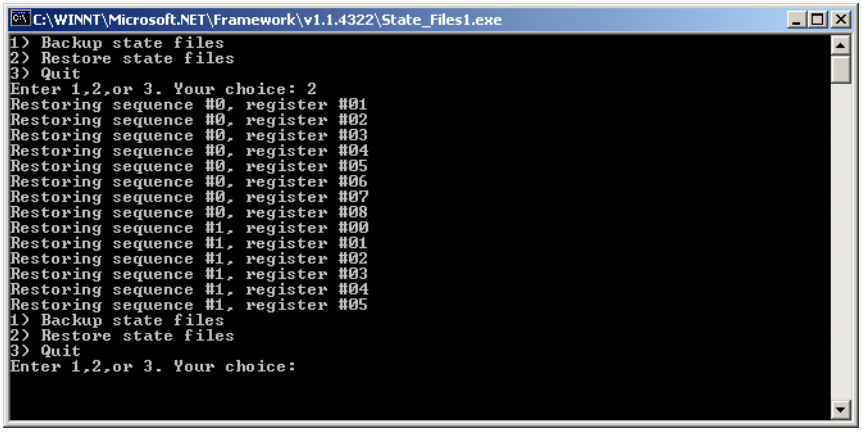
Save and Recall Programming Example Using VISA and C#

The following programming example uses VISA and C# to save and recall signal generator instrument states. Instruments states are saved to and recalled from your computer. This console program prompts the user for an action: Backup State Files, Restore State Files, or Quit.

The Backup State Files choice reads the signal generator's state files and stores it on your computer in the same directory where the State_Files.exe program is located. The Restore State Files selection downloads instrument state files, stored on your computer, to the signal generator's State directory. The Quit selection exists the program. The figure below shows the console interface and the results obtained after selecting the Restore State Files operation.

The program uses VISA library functions. Refer to the Agilent VISA User's Manual available on Agilent's website: *http:\\www.agilent.com* for more information on VISA functions.

The program listing for the State_Files.cs program is shown below. It is available on the CD–ROM in the programming examples section under the same name.


C:\WINNT\Microsoft.NET\Framework\v1.1.4322\State_Files1.exe
1) Backup state files
2) Restore state files
3) Quit
Enter 1,2,or 3. Your choice: 2
Restoring sequence #0, register #01
Restoring sequence #0, register #02
Restoring sequence #0, register #03
Restoring sequence #0, register #04
Restoring sequence #0, register #05
Restoring sequence #0, register #06
Restoring sequence #0, register #07
Restoring sequence #0, register #08
Restoring sequence #1, register #00
Restoring sequence #1, register #01
Restoring sequence #1, register #02
Restoring sequence #1, register #03
Restoring sequence #1, register #04
Restoring sequence #1, register #05
1) Backup state files
2) Restore state files
3) Quit
Enter 1,2,or 3. Your choice:

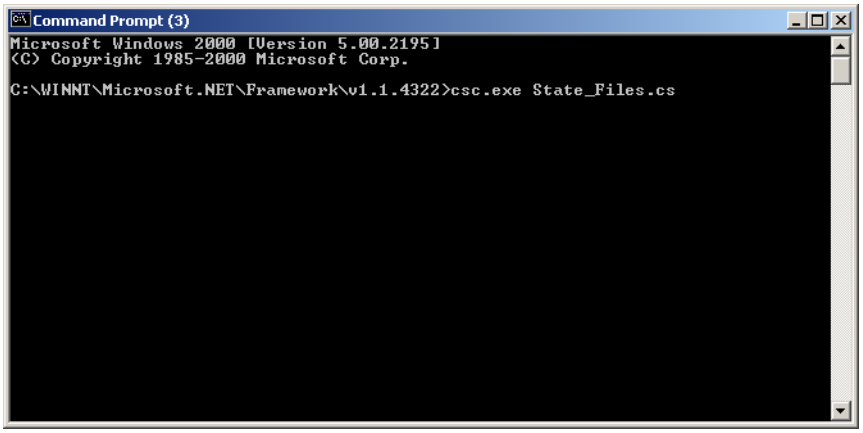
C# and Microsoft .NET Framework

The Microsoft .NET Framework is a platform for creating Web Services and applications. There are three components of the .NET Framework: the common language runtime, class libraries, and Active Server Pages, called ASP.NET. Refer to the Microsoft website for more information on the .NET Framework.

The .NET Framework must be installed on your computer before you can run the State_Files program. The framework can be downloaded from the Microsoft website and then installed on your computer.

Perform the following steps to run the State_Files program.

1. Copy the State_Files.cs file from the CD–ROM programming examples section to the directory where the .NET Framework is installed.
2. Change the TCPIP0 address in the program from TCPIP0::000.000.000.000 to your signal generator's address.
3. Save the file using the .cs file name extension.
4. Run the Command Prompt program. Start > Run > "cmd.exe". Change the directory for the command prompt to the location where the .NET Framework was installed.
5. Type csc.exe State_Files.cs at the command prompt and then press the Enter key on the keyboard to run the program. The following figure shows the command prompt interface.



The State_Files.cs program is listed below. You can copy this program from the examples directory on the signal generator's CD–ROM.

NOTE The *State_Files.cs* example uses the ESG in the programming code but can be used with the PSG or Agilent MXG.

```
//*****************************************************************************************
// FileName: State_Files.cs
//
// This C# example code saves and recalls signal generator instrument states. The saved
// instrument state files are written to the local computer directory computer where the
// State_Files.exe is located. This is a console application that uses DLL importing to
// allow for calls to the unmanaged Agilent IO Library VISA DLL.
//
```

```
// The Agilent VISA library must be installed on your computer for this example to run.
// Important: Replace the visaOpenString with the IP address for your signal generator.
//
//*************************************************************************************
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;
using System.Collections;
using System.Text.RegularExpressions;

namespace State_Files

{
    class MainApp
    {
        // Replace the visaOpenString variable with your instrument's address.

        static public string visaOpenString = "TCPIP0::000.000.000.000"; //"GPIB0::19";
      //"TCPIP0::ESG3::INSTR";


public const uint DEFAULT_TIMEOUT = 30 * 1000;// Instrument timeout 30 seconds.
        public const int MAX_READ_DEVICE_STRING = 1024; // Buffer for string data reads.
      public const int TRANSFER_BLOCK_SIZE = 4096;// Buffer for byte data.

        // The main entry point for the application.

        [STAThread]

static void Main(string[] args)
        {

    uint defaultRM;// Open the default VISA resource manager
    if (VisaInterop.OpenDefaultRM(out defaultRM) == 0) // If no errors, proceed.
      {
         uint device;
         // Open the specified VISA device: the signal generator
     if (VisaInterop.Open(defaultRM, visaOpenString,VisaAccessMode.NoLock,
                          DEFAULT_TIMEOUT, out device) == 0)
        // if no errors proceed.
       {
```

```
bool quit = false;
  while (!quit)// Get user input
   {
   Console.Write("1) Backup state files\n" +
                 "2) Restore state files\n" +
                 "3) Quit\nEnter 1,2,or 3. Your choice: ");
      string choice = Console.ReadLine();
    switch (choice)
         {
          case "1":
          {
        BackupInstrumentState(device);  // Write instrument state
        break;                   // files to the computer
          }
                    case "2":
          {
         RestoreInstrumentState(device); // Read instrument state
         break;// files to the sig gen
          }
       case "3":
          {
        quit = true;
        break;
          }
        default:
          {
        break;
          }
        }
      }
     VisaInterop.Close(device);// Close the device
      }
     else
      {
     Console.WriteLine("Unable to open " + visaOpenString);
    }
        VisaInterop.Close(defaultRM);   // Close the default resource manager
    }
     else
      {
   Console.WriteLine("Unable to open the VISA resource manager");
    }
   }

 /* This method restores all the sequence/register state files located in
```

```
         the local directory (identified by a ".STA" file name extension)
         to the signal generator.*/

static public void RestoreInstrumentState(uint device)
   {
    DirectoryInfo di = new DirectoryInfo(".");// Instantiate object class
    FileInfo[] rgFiles = di.GetFiles("*.STA");  // Get the state files
    foreach(FileInfo fi in rgFiles)
      {
      Match m = Regex.Match(fi.Name, @"^(\d)_(\d\d)");
      if (m.Success)
        {
        string sequence = m.Groups[1].ToString();
        string register = m.Groups[2].ToString();
        Console.WriteLine("Restoring sequence #" + sequence +
                                    ", register #" + register);

/* Save the target instrument's current state to the specified sequence/
register pair. This ensures the index file has an entry for the specified
sequence/register pair. This workaround will not be necessary in future
revisions of firmware.*/

        WriteDevice(device,"*SAV " + register + ", " + sequence + "\n",
                            true); // << on SAME line!
        // Overwrite the newly created state file with the state
        // file that is being restored.
        WriteDevice(device, "MEM:DATA \"/USER/STATE/" + m.ToString() + "\",",
                            false); // << on SAME line!
        WriteFileBlock(device, fi.Name);
        WriteDevice(device, "\n", true);
                 }
              }
        }

/* This method reads out all the sequence/register state files from the signal
generator and stores them in your computer's local directory with a ".STA"
extension */

static public void BackupInstrumentState(uint device)
   {
   // Get the memory catalog for the state directory
      WriteDevice(device, "MEM:CAT:STAT?\n", false);
```

```
      string catalog = ReadDevice(device);
      /* Match the catalog listing for state files which are named
         (sequence#)_(register#)  e.g.  0_01, 1_01, 2_05*/
      Match m = Regex.Match(catalog, "\"(\\d_\\d\\d),");
      while (m.Success)
         {
        // Grab the matched filename from the regular expresssion
        string nextFile = m.Groups[1].ToString();
       // Retrieve the file and store with a .STA extension
       // in the current directory
       Console.WriteLine("Retrieving state file: " + nextFile);
         WriteDevice(device, "MEM:DATA? \"/USER/STATE/" + nextFile + "\"\n", true);
      ReadFileBlock(device, nextFile + ".STA");
      // Clear newline
      ReadDevice(device);
      // Advance to next match in catalog string
      m = m.NextMatch();
       }
   }

/*  This method writes an ASCII text string (SCPI command) to the signal generator.
If the bool "sendEnd" is true, the END line character will be sent at the
conclusion of the write. If "sendEnd is false the END line will not be sent.*/

static public void WriteDevice(uint device, string scpiCmd, bool sendEnd)
   {
    byte[] buf = Encoding.ASCII.GetBytes(scpiCmd);
    if (!sendEnd) // Do not send the END line character
      {
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
      }
    uint retCount;
    VisaInterop.Write(device, buf, (uint)buf.Length, out retCount);
    if (!sendEnd) // Set the bool sendEnd true.
      {
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
      }
   }

// This method reads an ASCII string from the specified device
static public string ReadDevice(uint device)
   {
```

```
    string retValue = "";
    byte[] buf = new byte[MAX_READ_DEVICE_STRING]; // 1024 bytes maximum read
    uint retCount;
    if (VisaInterop.Read(device, buf, (uint)buf.Length -1, out retCount) == 0)
      {
      retValue = Encoding.ASCII.GetString(buf, 0, (int)retCount);
      }
    return retValue;
   }

/* The following method reads a SCPI definite block from the signal generator
and writes the contents to a file on your computer. The trailing
newline character is NOT consumed by the read.*/

static public void ReadFileBlock(uint device, string fileName)
   {
    // Create the new, empty data file.
    FileStream fs = new FileStream(fileName, FileMode.Create);
    // Read the definite block header:  #{lengthDataLength}{dataLength}
    uint retCount = 0;
    byte[] buf = new byte[10];
    VisaInterop.Read(device, buf, 2, out retCount);
    VisaInterop.Read(device, buf, (uint)(buf[1]-'0'), out retCount);
     uint fileSize = UInt32.Parse(Encoding.ASCII.GetString(buf, 0, (int)retCount));
    // Read the file block from the signal generator
    byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
    uint bytesRemaining = fileSize;

    while (bytesRemaining != 0)
      {
       uint bytesToRead = (bytesRemaining < TRANSFER_BLOCK_SIZE) ?
       bytesRemaining : TRANSFER_BLOCK_SIZE;
       VisaInterop.Read(device, readBuf, bytesToRead, out retCount);
       fs.Write(readBuf, 0, (int)retCount);
       bytesRemaining -= retCount;
      }
    // Done with file
    fs.Close();
   }

/* The following method writes the contents of the specified file to the
specified file in the form of a SCPI definite block.  A newline is
```

```
NOT appended to the block and END is not sent at the conclusion of the
write.*/

static public void WriteFileBlock(uint device, string fileName)
   {
    // Make sure that the file exists, otherwise sends a null block
    if (File.Exists(fileName))
     {
     FileStream fs = new FileStream(fileName, FileMode.Open);
     // Send the definite block header: #{lengthDataLength}{dataLength}
     string fileSize = fs.Length.ToString();
     string fileSizeLength = fileSize.Length.ToString();
     WriteDevice(device, "#" + fileSizeLength + fileSize, false);
     // Don't set END at the end of writes
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 0);
     // Write the file block to the signal generator
     byte[] readBuf = new byte[TRANSFER_BLOCK_SIZE];
     int numRead = 0;
     uint retCount = 0;
     while ((numRead = fs.Read(readBuf, 0, TRANSFER_BLOCK_SIZE)) != 0)
        {
        VisaInterop.Write(device, readBuf, (uint)numRead, out retCount);
        }
       // Go ahead and set END on writes
     VisaInterop.SetAttribute(device, VisaAttribute.SendEndEnable, 1);
     // Done with file
     fs.Close();
     }
     else
     {
     // Send an empty definite block
     WriteDevice(device, "#10", false);
     }
    }
   }

// Declaration of VISA device access constants
public enum VisaAccessMode
    {
        NoLock = 0,
        ExclusiveLock = 1,
        SharedLock = 2,
```

```
        LoadConfig = 4
    }

// Declaration of VISA attribute constants
public enum VisaAttribute
    {
        SendEndEnable = 0x3FFF0016,
        TimeoutValue  = 0x3FFF001A
    }

// This class provides a way to call the unmanaged Agilent IO Library VISA C
// functions from the C# application

public class VisaInterop
    {
        [DllImport("agvisa32.dll", EntryPoint="viClear")]
        public static extern int Clear(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viClose")]
        public static extern int Close(uint session);

        [DllImport("agvisa32.dll", EntryPoint="viFindNext")]
        public static extern int FindNext(uint findList, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viFindRsrc")]
        public static extern int FindRsrc(
            uint session,
            string expr,
            out uint findList,
            out uint retCnt,
            byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viGetAttribute")]
public static extern int GetAttribute(uint vi, VisaAttribute attribute, out uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viOpen")]
        public static extern int Open(
            uint session,
            string rsrcName,
            VisaAccessMode accessMode,
            uint timeout,
            out uint vi);
```

```
        [DllImport("agvisa32.dll", EntryPoint="viOpenDefaultRM")]
        public static extern int OpenDefaultRM(out uint session);

        [DllImport("agvisa32.dll", EntryPoint="viRead")]
        public static extern int Read(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);

        [DllImport("agvisa32.dll", EntryPoint="viSetAttribute")]
public static extern int SetAttribute(uint vi, VisaAttribute attribute, uint attrState);

        [DllImport("agvisa32.dll", EntryPoint="viStatusDesc")]
        public static extern int StatusDesc(uint vi, int status, byte[] desc);

        [DllImport("agvisa32.dll", EntryPoint="viWrite")]
        public static extern int Write(
            uint session,
            byte[] buf,
            uint count,
            out uint retCount);
    }
}
```

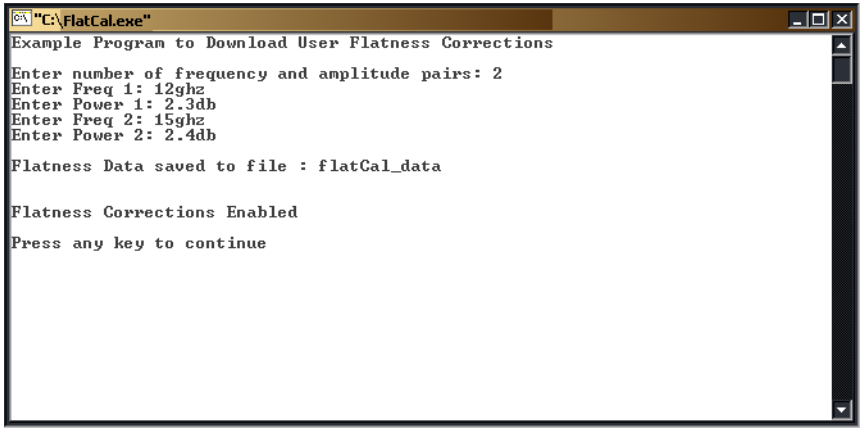
User Flatness Correction Downloads Using C++ and VISA

This sample program uses C++ and the VISA libraries to download user–flatness correction values to the signal generator. The program uses the LAN interface but can be adapted to use the GPIB interface by changing the address string in the program.

You must include header files and resource files for library functions needed to run this program. Refer to "Running C++ Programs" on page 70 for more information.

The FlatCal program asks the user to enter a number of frequency and amplitude pairs. Frequency and amplitude values are entered via the keyboard and displayed on the console interface. The values are then downloaded to the signal generator and stored to a file named flatCal_data. The file is then loaded into the signal generator's memory catalog and corrections are turned on. The figure below shows the console interface and several frequency and amplitude values. Use the same format, shown in the figure below, for entering frequency and amplitude pairs (for example, `12ghz`, `1.2db`).

Figure 7-3 FlatCal Console Application


"C:\FlatCal.exe"
Example Program to Download User Flatness Corrections
Enter number of frequency and amplitude pairs: 2
Enter Freq 1: 12ghz
Enter Power 1: 2.3db
Enter Freq 2: 15ghz
Enter Power 2: 2.4db
Flatness Data saved to file : flatCal_data
Flatness Corrections Enabled
Press any key to continue

The program uses VISA library functions. The non–formatted viWrite VISA function is used to output data to the signal generator. Refer to the Agilent VISA User's Manual available on Agilent's website: *http:\\www.agilent.com* for more information on VISA functions.

The program listing for the FlatCal program is shown below. It is available on the CD–ROM in the programming examples section as flatcal.cpp.

```
//*************************************************************************************
// PROGRAM NAME:FlatCal.cpp
//
// PROGRAM DESCRIPTION:C++ Console application to input frequency and amplitude
// pairs and then download them to the signal generator.
//
// NOTE: You must have the Agilent IO Libraries installed to run this program.
//
// This example uses the LAN/TCPIP interface to download frequency and amplitude
// correction pairs to the signal generator. The program asks the operator to enter
// the number of pairs and allocates a pointer array listPairs[] sized to the number
// of pairs.The array is filled with frequency nextFreq[] and amplitude nextPower[]
// values entered from the keyboard.
//
//*************************************************************************************
// IMPORTANT: Replace the 000.000.000.000 IP address in the instOpenString declaration
// in the code below with the IP address of your signal generator.
//*************************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include  "visa.h"
#include <string.h>

//   IMPORTANT:
//   Configure the following IP address correctly before compiling and running

char* instOpenString ="TCPIP0::000.000.000.000::INSTR";//your PSG's IP address

const int MAX_STRING_LENGTH=20;//length of frequency and power strings
const int BUFFER_SIZE=256;//length of SCPI command string

int main(int argc, char* argv[])
{
    ViSession defaultRM, vi;
    ViStatus status = 0;

    status = viOpenDefaultRM(&defaultRM);//open the default resource manager

    //TO DO: Error handling here

    status = viOpen(defaultRM, instOpenString, VI_NULL, VI_NULL, &vi);
```

```
 if (status)//if any errors then display the error and exit the program
 {
     fprintf(stderr, "viOpen failed (%s)\n", instOpenString);
     return -1;
 }

printf("Example Program to Download User Flatness Corrections\n\n");
printf("Enter number of frequency and amplitude pairs: ");
 int num = 0;

 scanf("%d", &num);

 if (num > 0)
 {
     int lenArray=num*2;//length of the pairsList[] array. This array
     //will hold the frequency and amplitude arrays

     char** pairsList = new char* [lenArray]; //pointer array

     for (int n=0; n < lenArray; n++)//initialize the pairsList array
                                     //pairsList[n]=0;

     for (int i=0; i < num; i++)
     {
       char* nextFreq = new char[MAX_STRING_LENGTH+1]; //frequency array
       char* nextPower = new char[MAX_STRING_LENGTH+1];//amplitude array
       //enter frequency and amplitude pairs i.e 10ghz .1db
       printf("Enter Freq %d: ", i+1);
       scanf("%s", nextFreq);
       printf("Enter Power %d: ",i+1);
       scanf("%s", nextPower);
       pairsList[2*i] = nextFreq;//frequency
       pairsList[2*i+1]=nextPower;//power correction
     }

 unsigned char str[256];//buffer used to hold SCPI command

 //initialize the signal generator's user flatness table
 sprintf((char*)str,":corr:flat:pres\n"); //write to buffer
 viWrite(vi, str,strlen((char*str),0);    //write to PSG
 char c = ',';//comma separator for SCPI command
 for (int j=0; j< num; j++)   //download pairs to the PSG
```

```
      {
       sprintf((char*)str,":corr:flat:pair %s %c %s\n",pairsList[2*j], c,
              pairsList[2*j+1]); // << on SAME line!
       viWrite(vi, str,strlen((char*)str),0);
      }
    //store the downloaded correction pairs to PSG memory
    const char* fileName = "flatCal_data";//user flatness file name
    //write the SCPI command to the buffer str
     sprintf((char*)str, ":corr:flat:store \"%s\"\n", fileName);//write to buffer
    viWrite(vi,str,strlen((char*)str),0);//write the command to the PSG
    printf("\nFlatness Data saved to file : %s\n\n", fileName);

   //load corrections
   sprintf((char*)str,":corr:flat:load \"%s\"\n", fileName); //write to buffer
   viWrite(vi,str,strlen((char*)str),0); //write command to the PSG
   //turn on corrections
   sprintf((char*)str, ":corr on\n");
   viWrite(vi,str,strlen((char*)str),0");
   printf("\nFlatness Corrections Enabled\n\n");
   for (int k=0; k< lenArray; k++)
     {
      delete [] pairsList[k];//free up memory
     }
      delete [] pairsList;//free up memory
    }

    viClose(vi);//close the sessions
    viClose(defaultRM);

    return 0;
}
```

Data Transfer Troubleshooting (N5162A, N5182A, E4438C and E8267D Only)

NOTE The section, User FIR Filter Coefficient File Download Problems, applies to the N5162A and N5182A with Option 651, 652, or 654; the E4438C with Option 001, 002, 601, or 602; and the E8267D with Option 601 or 602.

The remaining sections, User File Download Problems and PRAM Download Problems, apply *only* to the E4438C with Option 001, 002, 601, or 602; and the E8267D with Option 601 or 602.

This section is divided by the following data transfer methods:

"User File Download Problems" on page 351

"PRAM Download Problems" on page 352

"User FIR Filter Coefficient File Download Problems" on page 354

Each section contains the following troubleshooting information:

- a list of symptoms and possible causes of typical problems encountered while downloading data to the signal generator
- reminders regarding special considerations and file requirements
- tips on creating data, transferring data, data application and memory usage

User File Download Problems

Table 7-15 Use–File Download Trouble – Symptoms and Causes

Symptom	Possible Cause
At the RF output, some data modulated, some data missing	Data does not completely fill an integer number of timeslots. If a user file fills the data fields of more than one timeslot in a continuously repeating framed transmission, the user file will be restarted after the last timeslot containing completely filled data fields. For example, if the user file contains enough data to fill the data fields of 3.5 timeslots, firmware will load 3 timeslots with data and restart the user file after the third timeslot. The last 0.5 timeslot worth of data will never be modulated.

Data Requirements

- The user file selected must entirely fill the data field of each timeslot.
- The user file must be a multiple of 8 bits, so that it can be represented in ASCII characters.
- Available volatile memory must be large enough to support both the data field bits and the framing bits.

Requirement for Continuous User File Data Transmission

"Integer Number of Timeslots" Requirement for Multiple–Timeslots

If a user file fills the data fields of more than one timeslot in a continuously repeating framed transmission, the user file is restarted after the last timeslot containing completely filled data fields. For example, if the user file contains enough data to fill the data fields of 3.5 timeslots, the firmware loads 3 timeslots with data and restart the user file after the third timeslot. The last 0.5 timeslot worth of data is never modulated.

To solve this problem, add or subtract bits from the user file until it completely fills an integer number of timeslots

"Multiple–of–8–Bits" Requirement

For downloads to bit and binary memory, user file data must be downloaded in multiples of 8 bits (bytes), since SCPI specifies data in bytes. Therefore, if the original data pattern's length is not a multiple of 8, you need to:

- add bits to complete the ASCII character
- replicate the data pattern to generate a continuously repeating pattern with no discontinuity
- truncate the excess bits

NOTE The "multiple–of–8–bits" data length requirement is in *addition* to the requirement of completely filling the data field of an integer number of timeslots.

Using Externally Generated, Real–Time Data for Large Files

When the data fields must be continuous data streams, and the size of the data exceeds the available PRAM, real–time data and synchronization can be supplied by an external data source to the front panel DATA, DATA CLOCK, and SYMBOL SYNC connectors. This data can be continuously transmitted, or can be framed by supplying a data–synchronous burst pulse to the EXT1 INPUT connector on the front panel. Additionally, the external data can be multiplexed into internally generated framing

PRAM Download Problems

Table 7-16 PRAM Download – Symptoms and Causes

Symptom	Possible Cause
The transmitted pattern is interspersed with random, unwanted data.	Pattern reset bit not set. Insure that the pattern reset bit (bit 7, value 128) is set on the last byte of your downloaded data.

Table 7-16 PRAM Download – Symptoms and Causes

Symptom	Possible Cause
ERROR –223, Too much data	PRAM download exceeds the size of PRAM memory. Either use a smaller pattern or get more memory by ordering the appropriate hardware option.

Data Requirements

- The signal generator requires a file with a minimum of 60 bytes
- For every data bit (bit 0), you must provide 7 bits of control information (bits 1–7).

Table 7-17 PRAM Data Byte

Bit	Function	Value	Comments
0	Data	0/1	This is the data (payload) bit. It is "unspecified" when burst (bit 2) is set to 0.
1	Reserved	0	Always 0
2	Burst	0/1	1 = RF on 0 = RF off For non–bursted, non–TDMA systems, to have a continuous signal, set this bit to 1 for all bytes. For framed data, set this bit to 1 for *on* timeslots and 0 for *off* timeslots.
3	Reserved	0	Always 0
4	Reserved	1	Always 1
5	Reserved	0	Always 0
6	EVENT1 Output	0/1	To have the signal generator output a single pulse at the EVENT 1 connector, set this bit to 1. Use this output for functions such as a triggering external hardware to indicate when the data pattern begins and restarts, or creating a data–synchronous pulse train by toggling this bit in alternate bytes.
7	Pattern Reset	0/1	0 = continue to next sequential memory address. 1 = end of memory and restart memory playback. This bit is set to 0 for all bytes except the last byte of PRAM. To restart the pattern, set the last byte of PRAM to 1.

User FIR Filter Coefficient File Download Problems

Table 7-18 User FIR File Download Trouble – Symptoms and Causes

Symptom	Possible Cause
ERROR –321, Out of memory	There is not enough memory available for the FIR coefficient file being downloaded. To solve the problem, either reduce the file size of the FIR file or delete unnecessary files from memory.
ERROR –223, Too much data	User FIR filter has too many symbols. Real–Time cannot use a filter that has more than 64 symbols (512 symbols maximum for Arb). You may have specified an incorrect oversample ratio in the filter table editor.

Data Requirements

- Data must be in ASCII format.
- Downloads must be in list format.
- Filters containing more symbols than the hardware allows (64 for real–time and 512 for Arb) will not be selectable for the configuration.

Symbols

.NET framework, 336

Numerics

2's complement data format, 205
8757d
  GPIB addresses, 110
  pass-thru commands, 109
  pass-thru programming, 110

A

abort function, 74
address
  GPIB address, 25
  IP address, 30
Agilent
  BASIC, *See* HP BASIC
  e8663b
    global settings, configuring, 19, 288
    memory allocation, non-volatile memory, 294
    memory allocation, volatile memory, 293
    Pulse/RF Blank, configuring, 288
    setting GPIB address, 26
    volatile memory types, 291
    web server, on, 12
  esg
    global settings, configuring, 19, 288
    memory allocation, non-volatile memory, 216, 294
    memory allocation, volatile memory, 293
    Pulse/RF Blank, configuring, 288
    setting GPIB address, 26
    volatile memory types, 291
    Waveform Download Assistant, 243
    web server, on, 12
  IO Libraries
    Suite, using interactive IO, 36
    version J, 39
    Version M, 8
    version M, 7, 36, 39, 67
    versions, earlier, 7
  mxg
    global settings, configuration, 18
    global settings, configuring, 288
    memory allocation, non-volatile memory, 215, 293, 294
    memory allocation, volatile memory, 293
    setting GPIB address, 25
    volatile memory types, 291
    Waveform Download Assistant, 243
    web server, on, 12
  psg
    global settings, configuring, 19, 288
    memory allocation, non-volatile memory, 216, 294
    memory allocation, volatile memory, 293
    Pulse/RF Blank, configuring, 288
    setting GPIB, 26
    volatile memory types, 291
    Waveform Download Assistant, 243
    web server, on, 12
  SICL, 9, 27, 74
  Signal Studio, 242
  Signal Studio Toolkit, 198
  VISA, 9, 27, 59, 67, 74
  VISA COM Resource Manager 1.0, 71
Agilent IO Libraries
  earlier, 7
  Suite, 6
Agilent IO Libraries Suite, 6
Agilent IO Library
  versions M (earlier), 37
Agilent IO Library Version 15.0
  VISA configuration, 37
Agilent IO Library Version M or ealier
  VISA configuration, 37
Agilent VISA, 9
ARB waveform file downloads
  data requirements
    waveform, 199
  download utilities, 198
  waveform download utilities, 242
ASCII, data, 77
Auto-IP, 33
AUXILIARY INTERFACE, *See* RS-232

B

baseband operation status group, registers, 175–177
Baseband Studio
  for Waveform Capture and Playback, 210
BASIC
  ABORT, 74
  CLEAR, 76
  ENTER, 78
  LOCAL, 76
  LOCAL LOCKOUT, 75
  OUTPUT, 77
  REMOTE, 75
  See HP BASIC
big-endian
  byte order, interleaving and byte swapping, 230
  changing byte order, 202
  example, programming, 275
binary
  data
    framed, 302
    unframed, 301
  file

Index

downloads commands, 310
modifying hex editor, 312
bit
file
downloads and commands, 309
modifying hex editor, 313
order, user file, 298
status, monitoring, 161
values, 160
bits and bytes, 199
byte order
byte swapping, 202
changing byte order, 202
interleaving I/Q data, 230

C

C
AC-coupled FM signals
generating externally applied, 91
CW signals, generating, 89
data questionable
status register, reading, 101
FM signals, generating internally applied, 93
reading the service request interrupt, 105
Sockets LAN, programming, 116
states, saving and recalling, 98
C and VISA
GPIB
queries, 87
GPIB, interface check, 80
C/C++, 10
C#
programming examples, 71
remote control, 10
VISA, example, 337
C++
programming examples, 70, 247
VISA, generating a step-swept signal, 95
C++ and VISA
generating a step-swept signal, 95
cable
USB, 67
carrier
activating, FIR filters, 335
modulating, FIR filters, 335
CDMA modulation
data, FIR filter, 335
Checking Available Memory, 295
clear
command, 76
function, 76
CLS command, 163
command
CLS, 163
format programming, user file data, 307
format user file, downloading, 306
prompt, 35, 139
window PC, using, 315
window UNIX, using, 315
commands
8757d
pass-thru, troubleshooting, 111
Agilent mxg, menu path, 17
downloads, binary file, 310
downloads, bit file, 309
e8663b, 18
e8663b, menu path, 18
esg, menu path, 18
GPIB, 74, 75, 76, 77, 78
pass-thru, 8757d, 109
psg, menu path, 18
computer interface, 4
computer-to-instrument communication
VISA
configuration, automatic, 7
VISA configuration, (manual), 7
condition registers, description, 169
Configuration
IO Libraries, 37
configuring, VXI-11, 39
connection expert, 6
connection wizard, 6
controller, 27
creating waveform data
C++, using, 227
saving to a text file for review, 230
creating waveform files
overview, 197
crossover cable, private LAN, 34
csc.exe, 336
custom
modulation data, FIR filter, 335
real-time, high data rates, 319
user file data, memory usage, 303

D

DAC input values, 202
data
binary, framed, 302
binary, unframed, 301
encryption, 219
format, e443xb signal generator, 244
requirements, waveform, 199
data questionable
See also data questionable registers
filters

BERT transition, 194
calibration transition, 191
frequency transition, 185
modulation transition, 188
power transition, 182
transition, 180
groups
BERT status, 193
calibration status, 190
frequency status, 184
modulation status, 187
power status, 181
status, 177
status register
reading, using VISA and C, 101
data questionable registers
BERT event, 195
BERT event enable, 195
BERT, condition, 194
calibration condition, 191
calibration event, 191
calibration event enable, 192
condition, 178
event, 180
event enable, 180
frequency condition, 185
frequency event, 185
frequency event enable, 186
modulation condition, 188
modulation event, 188
modulation event enable, 189
power condition, 182
power event, 182
power event enable, 183
data rates, high
custom, real-time, 319
data requirements, FIR filter downloads, 333
data types
binary, 290
bit, 290
defined, 290
FIR filter states, 290
PRAM, 290
user flatness correction, 290
decryption, 219
developing programs, 70
device, add, 8
DHCP, 11, 33
directory, root, 293
disclaimer
LXI-B, 1, 47
DNS, 35
documentation, list of, ix
DOS command prompt, 41
download
binary file data, 301
bit file data, 298
FIR filter coefficient data, 333
user file data
FTP procedures, 313
unencrypted files for extraction, 332
unencrypted files for no extraction, 332
user flatness, 336
utilities
Agilent Signal Studio, Toolkit, 198
IntuiLink for signal generators, 198
Waveform Download Assistant, 198
waveform data
advanced programming languages, 236
commands, 218
e443xb signal generator files, 203, 243
encrypted files for extraction, 223
encrypted files for no extraction, 221
FTP procedures, 224
memory locations, 219
overview, 197, 233
simulation software, 234
unencrypted files for extraction, 221
unencrypted files for no extraction, 221
user-data files, using, 289
download libraries, 9
downloaded PRAM files
data sources, 329
downloading
block data
SCPI command, 326
SCPI command, programming syntax, 327
C++, using, 247
HP Basic, 280
MATLAB, 271
Visual Basic, 277
downloads, PRAM data
e4438c, 320
e8267d, 320

E

e443xb
files
downloading, 243, 246
formatting, 203, 244
programming examples, 262
storing, 244
programming examples, 280
e8663b
See Agilent e8663b
edit VISA config, 8
EnableRemote, 75

Index

encryption
  downloading
    for extraction, 223
    for no extraction, 221
  extracting waveform data, 222, 223
  I/Q files, 219
  I/Q files, agilent mxg (only), 219
  securewave directory
    agilent mxg (only), 219
    esg, 219
    psg, 219
  waveform data, 218
enter function, 78
errors, 20, 36
ESE commands, 163
esg
  See Agilent esg
even number of samples, 209
event enable register
  description, 169
event registers
  description, 169
example programs *See* programming examples, 246
examples
  pass-thru commands, 109
  save and recall, 337
  Telnet, 45
external media
  See USB media
external memory
  See USB media
externally applied AC-coupled FM signals
  generate, using VISA and C, 91
extract user file data, 332
extracting
  PRAM files, 330

F

file size
  determining
    PRAM, 324
  minimum
    PRAM, 325
  PRAM, 324
file types
  See data types
files
  decryption, 219
  encryption, 218, 219
  encryption, agilent mxg (only), 219
  error messages, 20
  extraction commands and file paths, 220
  header information, 207, 219
  large, generating real-time data, 352
  PRAM, modifying, 332
  transfer methods, 219
  transferring, 45
  waveform download utilities, 242
  waveform structure, 207
filters
  See transition filters
FIR
  filter data
    CDMA modulation, 335
    custom modulation, 335
    TDMA format, 334
    W-CDMA modulation, 335
  filters
    carrier, activating, 335
    carrier, modulating, 335
    data limitations, 333
firmware
  loading older versions, caution, 1
firmware status, monitoring, 161
framed data, usage
  volatile memory, PRAM, 304
front panel
  USB
    connector, Type-A, 68
    flash memory sticks, 68
    media, 68
    USB media, 68
FTP
  commands for downloading and extracting files, 332
  downloading and extracting files, commands, 222–223
  internet explorer, using, 314
  methods, 219
  procedures for downloading files, 224, 313
  using, 45
  web server procedure, 226, 315

G

Getting Started Wizard, 26
global settings
  Agilent mxg, 18, 288
  e8663b, 19, 288
  esg, 19, 288
  psg, 19, 288
GPIB
  8757d, addresses, 110
  address, 25, 112
  Agilent mxg, setting address, 25
  configuration, 25
  controller, 27
  e8663b, setting address, 26
  esg, setting address, 26

interface, 4, 25
interface cards, 23, 73
IO libraries, 9
listener, 27
overview, 23, 73
program examples, 27, 74, 80, 87
SCPI commands, 27
talker, 27
troubleshooting, 26
using VISA and C, 80
verifying operation, 26
GPIB address
psg, setting address, 26

H

hardware
layers, remote programming, 2
status, monitoring, 161
help mode
setting
Agilent mxg, 18
e8663b, 19
esg, 19
psg, 19
hex editor
binary file, modifying, 312
bit file, modifying, 313
hexadecimal data, 275
hostname, 30, 112
hostname, setting, 34
Agilent mxg, 32
Agilent mxg menus, 30
DHCP LAN, e8663b, 34
DHCP LAN, esg, 34
DHCP LAN, psg, 34
DHCP/Auto I/P LAN, Agilent mxg, 33
e8663b, 32
e8663b menus, 31
esg/psg/e8663b, 32
esg/psg/e8663b menus, 31
HP BASIC, 10
HP Basic
I/O library, 59
local lockout, 81
programming examples, 280
queries, 84
RS-232
control, 59
queries, 65, 145
HyperTerminal, 62

I

I/O libraries
See IO libraries
I/Q data
creating, advanced programming languages, 227
encryption, 218, 219
encryption, agilent mxg (only), 219
interleaving
big endian and little endian, 230
byte swapping, 230
little endian, byte swapping, 230
waveform data, creating, 205
memory locations, 213, 232
saving to a text file for review, 230
scaling, 203
waveform structure, 209
iabort, 74
ibloc, 76
ibstop, 74
ibwrt, 77
iclear, 76
IEEE standard, 23, 73
igpibllo, 75
iloc, 76
input values, DAC, 202
instrument
communication, 8
state files
overview, 336
SCPI commands, recalling, 336
SCPI commands, saving, 336
instrument status, monitoring, 149
interactive IO, 6, 36
interface
cards, 23, 73
GPIB, 25
LAN, 4
RS-232, 4
USB (Agilent mxg only), 4
interleaving, *See* I/Q data, 205
internal
web server
FTP procedure, 314
internal storage
See storage
internally applied FM signals
generate, using VISA and C, 93
IntuiLink for signal generators, 242
IO Config
Agilent IO libraries Suite, 6
computer-to-instrument communication, 7
VISA, manual, 8
IO Configure
Using VISA Assistant, 37
IO interface, 8
IO libraries, 6

Index

Agilent, suite, 6
GPIB interface, installing, 23
GPIB, installing interface cards, 73
GPIB, selecting for, 9
GPIB, verifying, 26
interactive IO, using, 36
program languages, overview, 5
RS-232, selecting for, 59
signal generator, remote control, 2
suite, overview, 6
USB, selecting for, 67
VISA LAN, troubleshooting, 38
IP address
LAN interface, 30
LAN, assigning, 30
setting, 30, 31, 33, 34
setting Agilent mxg, 32
setting e8663b, 32
setting esg/psg/e8663b, 32
iremote, 75

J

JAVA, 72, 139
Java
example, 72, 139

K

kjkj, 215

L

LabView, 10
LAN
Auto-IP configuration, 33
config, 38
configuration
Agilent mxg, 32, 33
esg/psg/e8663b, 32
menu, Agilent mxg, 30
menu, esg/psg/e8663b, 31, 34
summary, Agilent mxg, 16
web server, 11
DHCP configuration, 33
establishing a connection, 234, 236
hostname, 30
interface, 4
IO libraries, 9
LXI
interface protocols, 29, 47
manual configuration, 31
overview, 29
private, 34
program examples, 72, 112, 139, 141
programming
using JAVA, 72, 139
queries using sockets, 119
sockets, 112
sockets LAN, 29
Telnet, 41
troubleshooting, 35
verifying operation, 34
VXI-11
examples, using, 112
interface protocols, 29
perl, using, 141
programming examples, LAN, 112
sockets, programming, 72, 139
LAN configuration
e8663b, 32
menu, e8663b, 31, 34
libraries
GPIB functionality, verifying, 26
GPIB I/O libraries, selecting, 9
IO, Agilent, 2, 5
RS-232, 59
selecting, for computer, 9
USB, 67
list format, downloading
SCPI command, 325
list, error messages, 20
listener, 27
little-endian
byte order, interleaving and byte swapping, 230
loading waveforms, 239
local
echo, telnet, 44
function, 76
local lockout
function, 75
HP Basic, using, 81
location user-data file type
binary, 295
LSB, 200
LSB and MSB, 298
LSB/MSB, 275
LXI
class B, mxg, 4, 11
LXI-B
disclaimer, 1, 47
LXI-B subsystem, 47, 48, 49, 50, 51, 52, 53
LXI-B synchronizing time, 47, 48, 49, 50, 51, 52, 53

M

manual operation, 75
marker file, 207, 219
MATLAB, 10
download utility, 243

downloading data, 234
programming examples, 268
programming, introduction, 10
media
external
flash memory sticks, 68
front panel USB, 68
non-volatile memory, Agilent mxg, 291
waveform memory, 212
internal
non-volatile memory, Agilent mxg, 291
waveform memory, 212
USB
non-volatile memory, Agilent mxg, 291
memory
See also media
allocation, 214, 293
checking, available, 295
defined, 212, 291
location user-data file type
available memory, checking, 295
bit, 295
FIR, 295
flatness, 295
instrument state, 295
PRAM, 295
locations, 212, 291
non-volatile (NVWFM), 219
signal generator, maximum, 295
size, 216, 294
volatile (WFM1), 219
volatile and non-volatile, 291
memory usage
user file data
custom, 303
TDMA, 303
Microsoft .NET Framework
overview, 337
Mini-B (5-pin)
Rear panel connector, 68
MSB, 200
MSB and LSB, 298
MS-DOS Command Prompt, 35, 41
multiple-of-8-bits requirement
user file data, 352
multiple-timeslots
integer number of timeslots, 352
mxg
LXI class B, 4, 11
See Agilent mxg

N

n5161a/62a/81a/82a/83a
Pulse/RF Blank configuring, 288
National Instruments
NI-488.2, 27, 74
VISA, 9, 27, 59, 67, 74
negative transition filter, description, 169
NI libraries
SICL
GPIB I/O libraries, selecting, 9
NI-488.2
EnableRemote, 75
functions, 9
GPIB I/O libraries, selecting, 9
iblcr, 76
ibloc, 76
ibrd, 78
ibstop, 74
ibwrt, 77
LAN I/O libraries, selecting, 9
queries using C++, 85
RS-232 I/O libraries, selecting, 59
SetRWLS, 75
USB I/O libraries, selecting, 67
VISA, 9, 59
non-volatile memory
available
SCPI query, 296
external media, Agilent mxg, 291
internal media, Agilent mxg, 291
internal storage, Agilent mxg, 291
memory allocation, 294
Agilent mxg, 215, 293
esg, 216
psg, 216
securewave directory, 219
USB media, 68
USB media, Agilent mxg, 291
waveform, 212

O

OPC commands, 163
output command, 77
output function, 77

P

pass-thru commands, 109
PC, 275
PCI-GPIB, 27, 74
PERL
example, 141
phase discontinuity
avoiding, 210
Baseband Studio, for Waveform Capture and Playback, 210

Index

samples, 211
waveform, 209
phase distortion, 209
ping
program, 34
responses, 36
playing waveforms, 239
polling method (status registers), 161
ports, 116
positive transition filter, description, 169
PRAM
as data sources, 329
bit positions, 322
byte patterns, 322
data extracting SCPI command, syntax, 331
downloads, problems, 352
e4438c, data downloads, 320
e8267d, data downloads, 320
file size, 324
minimum, 325
file size, determining, 324
files
command syntax, for restoring, 330
command syntax, for storing, 330
extracting, 330
modifying, 332
non-volatile memory, storing, 330
understanding, 321
volatile memory, restoring, 330
volatile memory
framed data, usage, 304
unframed data, usage, 304
waveform, viewing, 323
private LAN, using, 34
problems
user
file downloads, 351
FIR filter downloads, 354
programming
8757d, using pass-thru, 110
creating waveform data, 227
downloading waveform data, 233
little endian order, byte swapping, 230
user file data
command format, 307
programming examples
C#, 71, 337
C++, 70, 247
e443xb
files, 262
e443xb files, 280
HP Basic, 280
introduction, 246
MATLAB, 268
pass-thru commands, 109
RS-232, queries using VISA and C, 65, 146
RS-232, using VISA and C, 64, 143
using, 69
using GPIB, 27, 74, 80, 87
using LAN, 72, 112, 139, 141
using RS-232, 63, 142
Visual Basic, 275, 277
VXI-11, 112
psg
See Agilent psg
Pulse/RF Blank
e8663b, setting, 288
esg, setting, 288
n5161a/62a/81a/82a/83a, setting, 288
psg, setting, 288

Q

queries
HP Basic, using, 84
queue, error, 20

R

ramp sweep, using pass-thru commands, 109
real-time
data files, generating large, 352
TDMA
user files, 315
rear panel connector
Mini-B, 68
recall states, 336
register system overview, 149
data questionable
See also data questionable registers
registers
See also data questionable registers
See also status registers
baseband operation
condition, 176
event, 177
event enable, 177
condition, description, 169
e8663b overall system, 154, 155
esg overall system, 156, 157
mxg overall system, 152, 153
psg overall system, 158, 159
standard event
bits, 171
status, 171
status enable, 171
standard operation
condition, 173
event, 174

event enable, 174
status byte, 168
status groups, register type descriptions, 169
remote, 18
remote annunciator, 142
remote function
HP Basic, 75
setting
Agilent mxg, 17
e8663b, 18
esg, 18
psg, 18
setting, e8663b, 18
remote interface
programming, 2
RS-232, 58
USB, 66
remote programming
hardware layers, 2
software layers, 2
RS-232
address, 64, 142
baud rate, 60
cable, 61
configuration, 60
echo, setting, 60
format parameters, 63
HP Basic, using queries, 65, 145
interface, 60
interfaces, 4
IO libraries, 59
overview, 58
program examples, 63, 142
programming examples, queries using VISA and C, 65, 146
programming examples, using VISA and C, 64, 143
settings, baud rate, 64, 142
verifying operation, 62

S

samples
even number, 209
waveform, 209
save and recall, 336
scaling I/Q data, 203
SCPI
error queue, 20
file transfer methods, 219
GPIB, overview, 23
programming languages, common, 10
register model, 149
web server control, 11
SCPI command, programming syntax
block data, downloading, 327
SCPI command, syntax
PRAM files, extracting, 331
SCPI commands
block data, downloading, 326
command line structure, 220
download e443xb files, 246
encrypted files, 221, 223
extraction, 218, 220, 221, 223, 332
for status registers
IEEE 488.2 common commands, 163
GPIB function statements, 27
instrument state files, recalling, 336
instrument state files, saving, 336
list format, downloading, 325
no extraction, 220, 221
unencrypted files, 221, 332
user FIR file downloads
sample command line, 334
securewave directory
decryption, file, 219
downloading encrypted files, 223
downloads, file, 219
encryption, file, 219
extracting waveform data, 222, 223
extraction, file, 219
sequences
waveforms, building, 241
service request
interrupt
reading, using VISA and C, 105
method
status registers, 162
using, 162
SetRWLS, 75
setting
help mode
Agilent mxg, 18
e8663b, 19
esg, 19
psg, 19
Pulse/RF Blank
e8663b, 288
esg, 288
n5161a/62a/81a/82a/83a, 288
psg, 288
SICL, 9, 59, 67
GPIB examples, 27, 74
iabort, 74
iclear, 76
igpibllo, 75
iloc, 76
iprintf, 77
iremote, 75

Index

iscanf, 78
NI libraries, 9
USB, using, 67
VXI-11, programming, 113
signal generator
monitoring status, 149
volatile memory types, 291
Waveform Download Assistant, 243
Signal Studio Toolkit, 198, 242
simulation software, 234
sockets
example, 116, 119
Java, 72, 139
LAN, 40, 112, 116
PERL, 141
UNIX, 116
Windows, 117
software
layers, remote programming, 2
libraries, IO, 6
SRE commands, 163
SRQ command, 162
SRQ method, status registers, 162
standard event status
enable register, 171
group, 170
register, 171
standard operation
condition register, 173
event enable register, 174
event register, 174
transition filters, 174
state files, 336
states
saving and recalling, using VISA and C, 98
status byte
e8663b overall register system, 154, 155
esg overall register system, 156, 157
group, 167
mxg overall register system, 152, 153
psg overall register system, 158, 159
register, 168
status groups
baseband operation, 175–177
data questionable
BERT, 193
calibration, 190
frequency, 184
modulation, 187
overview, 177
power, 181
registers, 169
standard
event, 170
status byte, 167
status registers
See also registers
accessing information, 161
bit values, 160
esg overall system, 157
hierarchy, 149
in status groups, 169
monitoring, 161
mxg overall system, 152, 153
overall system, 156
programming, 149
SCPI commands, 163
SCPI model, 149
setting and querying, 163
system overview, 149
using, 160
STB command, 163
storage
internal
non-volatile memory, Agilent mxg, 291
system requirements, 70

T

talker, 27
TCP/IP, 11
TCPIP, 8, 112
TDMA
data, FIR filter, 334
user file data, memory usage, 303
Telnet
DOS command prompt, 41
example, 45
PC, 42
UNIX, 44, 45
using, 41
Windows 2000, 43
Windows XP, 43
timeslots, integer number of
multiple-timeslots requirement, 352
Toolkit, Signal Studio, 198, 242
transition filters
baseband operation, 176
data questionable
BERT, 194
modulation, 188
negative and positive, 180
power, 182
data questionable calibration, 191
data questionable frequency, 185
description, 169
negative transition, description, 169
positive transition, description, 169

standard operation, 174
troubleshooting
8757d, pass-thru commands, 111
GPIB, 26
LAN, 35
ping
response errors, 36
PRAM downloads, 352
RS-232, 63
USB, 68
user file downloads, 351
user FIR filter downloads, 354
VISA assistant, 38
Type-A front panel USB connector, 68

U

unencrypted files
downloading for extraction, 221, 332
downloading for no extraction, 221, 332
extracting I/Q data, 331
unframed data, usage
volatile memory, PRAM, 304
USB
cable, 67
functionality, verification, 68
interface, 4
IO libraries, 67
setting up, 67
using, Agilent mxg, 66
verifying operation, 68
usb media
file extensions, 215
user data
file, modifying, 312
files, creating, 289
files, downloading, 289
memory, 291
root directory, 293
user file data, continuous transmission
requirements, 352
user files
bit order, 298
bit order, LSB and MSB, 298
data
binary, downloads, 297
bit, downloads, 297
multiple-of-8-bits requirement, 352
downloading
as the data source, 329
carrier, activating, 330
carrier, modulating, 330
command format, 306
modulating and activating the carrier, 312
selecting the user file as the data source, 311
framed transmissions, understanding, 315
real-time TDMA, 315
size, 302
user FIR file downloads
non-volatile memory, 334
selecting a downloaded user FIR file, 334
user flatness, 336
user-data file type
binary, memory location, 295
bit, memory location, 295
FIR, memory location, 295
flatness, memory location, 295
instrument state, memory location, 295
memory location, 295
PRAM, memory location, 295
user-data files
See user data
Using Connection Expert
configuring and running, 37

V

verifying waveforms, 239
Version M
IO Libraries, Agilent, 6, 7
viPrintf, 77
VISA, 9, 37, 59, 67
C++, generating a step-swept signal, 95
COM IO Library, 71
computer-to-instrument communication, 7
configuration
automatic, 8
manual, 8
CW signals, generating, 89
data questionable status register, reading, 101
FM signals, generating internally applied, 93
generating externally applied AC-coupled FM signals, 91
I/O libraries, 9
LAN client, 36
LAN, using, 9
library, 27, 74, 275
NI-488.2, 9
RS-232, using, 59
scanf, 78
service request interrupt, reading, 105
states, saving and recalling, 98
USB, using, 67
viPrintf, 77
Visual C++, generating a swept signal, 96
viTerminate, 74
VXI-11, 112
CW signals

Index

See VISA and C
VISA and C
CW signals, generating, 89
GPIB
interface check for, 80
queries, 87
VISA Assistant
GPIB functionality, verifying, 26
IO Config, 8
IO, Using interactive, 36
troubleshooting, 38
verifying instrument communication, 36
Visual Basic
IDE, 71
programming examples, 275
programming language, 10
references, 71
Visual C++
NI-488.2, queries using, 85
VISA, generating a swept signal, 96
Visual C++ and VISA
generating a swept signal, 96
viTerminate, 74
volatile memory
file, decryption, 219
file, encryption, 219
memory allocation, 214
Agilent e8663b, 293
Agilent esg, 293
Agilent psg, 293
securewave directory, 219
memory, volatile (WFM1), 219
signal generator, 291
types, signal generators, 291
waveform, 212
volatile memory available, SCPI query, 296
VXI-11, 112
configuration, 39
programming, 112
programming interface examples, 112
SICL, using, 113
using, 39
VISA, using, 114

W

waveform data
2's complement data format, 205
bits and bytes, 199
byte order, 202
byte swapping, 202
commands for downloading and extracting, 218–226, 306–315
creating, 227
DAC input values, 202
data requirements, 199
encrypted data, 215
encryption, 218–223
explained, 199
extracting, 218, 221–222
I and Q interleaving, 205
LSB and MSB, 200
saving to a text file for review, 230
waveform download
utilities
differences, 242
waveform downloads
advanced programming languages, using, 236
download utilities, using, 242
HP BASIC, using, 280–285
memory, 212
allocation, 214, 293
size, 216, 294
volatile and non-volatile, 212
samples, 209
simulation software, using, 234
structure, 209
troubleshooting files, 287
using advanced programming languages, 236
with Visual Basic 6.0, 277
waveform files
creating, 197
downloading, 197
waveform generation
C++, 247
HP Basic, using, 280
MATLAB, using, 268
Visual Basic 6.0, using, 275
waveforms
loading, 239
playing, 239
sequences, building, 241
verifying, 239
viewing, PRAM, 323
W-CDMA modulation data, FIR filter
See FIR
web server
Agilent
mxg, 12
communicating with, 11
e8663b, 12
esg, 12
internal, 314
psg, 12
Windows
2000, 43
98, 6
ME, 6

NT, 7
XP, 43
Windows ME, 6
Windows NT, 6
WriteIEEEBlock, 277

Index